

Porting Applications to the IRIS-4D Family

Porting Applications to the IRIS-4D Family

IRIS-4D Series



SiliconGraphics
Computer Systems

Document number: 007-0909-010

Porting Applications to the IRIS-4D Family

Version 1.0

Document Number 007-0909-010

Technical Publications:

Marcia Allen
Gail Kesner
Amy Smith
Diane Wilford

Engineering and Technical Marketing:

Kurt Akeley
Tom Davis
Gary Griffin
Todd Nordland
Mike Schulman
Thant Tessman
Chris Wagner
Mason Woo

© Copyright 1988, Silicon Graphics, Inc. - All rights reserved

This document contains proprietary and confidential information of Silicon Graphics, Inc., and is protected by Federal copyright law. The contents of this document may not be disclosed to third parties, copied or duplicated in any form, in whole or in part, without the express written permission of Silicon Graphics, Inc.

U.S. Government Limited Rights

Use, duplication or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (b) (2) of the Rights in Technical Data and Computer Software clause at 52.227-7013. Contractor/manufacturer is Silicon Graphics Inc., 2011 Stierlin Road, Mountain View, CA 94039-7311.

Porting Applications to the IRIS-4D Family

Version 1.0

Document Number 007-0909-010

Silicon Graphics, Inc.
Mountain View, California

The words IRIS, Geometry Link, Geometry Partners, Geometry Engine and Geometry Accelerator are trademarks of Silicon Graphics, Inc.

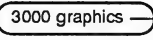
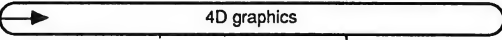
UNIX is a trademark of AT&T Bell Laboratories.

1. Introduction

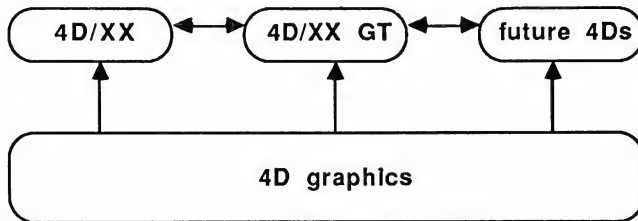
This book shows you how to port your application to the IRIS 4D family of workstations. Most of the information is directed toward people who are porting from an IRIS Series 2000 or 3000 workstation running GL2-W2.5/3.5 or 3.6. However, regardless of your current platform, this book provides useful porting information.

1.1 Developing a Porting Strategy

All IRIS-4D Series workstations run the same release of system software from Silicon Graphics. This means that UNIX, languages, and compilers are completely compatible across the 4D family; however, the IRIS 2000 and 3000 workstations cannot run this same release. This means that a port from an IRIS Series 2000/3000 to an IRIS-4D Series workstation involves changes in UNIX, languages, compilers, and graphics. In other words, this port involves both moving from software release IRIS3.6 to 4D3.0, and moving from 2000/3000 graphics to 4D graphics. This matrix illustrates these two paths; *yes* indicates that the software release can run on the workstation, *no* indicates that it cannot.

IRIS sys s/w	3XXX 	4D/XX 	4D/XX GT	future 4Ds
3.6	yes	no	no	no
4D2.0	no	yes	no	no
4D3.0	no	yes	yes	no
future 4D	no	yes	yes	yes

Once you have ported to the 4D family, however, subsequent ports between IRIS-4D Series workstations are much simpler and, at most, involve only changes to graphics.



Because some Graphics Library subroutines are dependent on the underlying hardware, there are some graphics differences between IRIS-4D Series workstations that affect the portability of your code. You can ensure that your application is completely portable from one 4D to another in two ways:

- Use only the Graphics Library subroutines that are common to all members of the family.
- Structure your code so it can use different subroutines depending on which 4D it is currently using.

If you use the first approach, the same version of code will run on all machines without modification. However, in most cases, the Graphics Library subroutines that are specific to a particular IRIS 4D are those subroutines that greatly increase graphics performance or take advantage of unique features. If you use the second approach, your code will be completely portable, and will also take advantage of the unique features of each machine.

This book shows you how to port your application so that it will run on any IRIS-4D Series workstation; it does not show you how to take advantage of any features or subroutines that not all members of the 4D family offer. Once you port your application according to the guidelines in this book, *Porting Applications Between IRIS-4D Series Workstations* summarizes the differences between members of the 4D family and shows you how to optimize your code for a particular IRIS 4D.

1.2 How This Document is Organized

This book is divided into two sections. The Section 1, “IRIS 3000 to 4D Compatibility Guide”, describes compatibility issues in the areas of languages and compilers, graphics, the operating system, networking, and hardware. It is a reference book that lists differences and modifications, and gives suggestions about how the differences affect porting. It also provides several tips for troubleshooting the port.

Section 2, “IRIS 3000 to 4D Conversion Tutorial”, concentrates on the graphics portion of your port. It is a step-by-step guide that shows you how the differences affect your port. It presents this information through both textual descriptions and sample programs.

Section 1: IRIS 3000 to 4D Compatibility Guide

Contents

1. Introduction	1-1
2. Porting C Code	2-1
2.1 Command Line Switches	2-1
2.1.1 Supported <i>cc</i> Switches	2-1
2.1.2 Supported <i>ld</i> Switches	2-2
2.1.3 Unsupported <i>cc</i> Switches	2-3
2.1.4 Unsupported <i>ld</i> Switches	2-4
2.2 Code Compatibility	2-4
2.2.1 Single and Double Precision	2-5
2.2.2 Trigonometric Functions	2-6
2.2.3 Unsigned Characters	2-6
2.2.4 C Data Files	2-6
2.2.5 The BSD Compatible Library	2-7
2.2.6 Miscellaneous Code Differences	2-8
2.2.7 New Reserved Words	2-11
2.2.8 Profiling	2-11
2.2.9 Standard Libraries	2-12
2.2.10 Compiling Large Programs	2-13
2.3 Making Libraries	2-14
3. Porting FORTRAN Code	3-1
3.1 Command Line Switches	3-1
3.1.1 Supported <i>f77</i> Switches	3-1
3.1.2 Unsupported <i>f77</i> Switches	3-2
3.1.3 Supported <i>ld</i> Switches	3-3
3.1.4 Unsupported <i>ld</i> Switches	3-4
3.2 Code Compatibility	3-5
3.2.1 Compiler Directives	3-5
3.2.2 Functions and Subroutines	3-6
3.2.3 I/O Compatibility	3-7
3.2.4 Non ANSI Standard Code	3-9

3.2.5	Miscellaneous Code Differences	3-9
3.2.6	Compiler Memory-Alignment Extensions	3-11
3.2.7	Compiling Large Programs	3-11
3.3	Making Libraries	3-12
4.	Interlanguage Calls	4-1
5.	Graphics Compatibility	5-1
5.1	The Window Manager	5-2
5.2	Screen Resolution	5-3
5.3	New Drawing Subroutines	5-3
5.4	Gouraud Shading	5-5
5.5	Drawing Modes	5-5
5.6	Overlays and Underlays	5-6
5.7	Cursors	5-7
5.8	Display Modes	5-8
5.9	Concave Polygons	5-10
5.10	Feedback Parsing	5-11
5.11	Textports and <i>wsh</i>	5-12
5.12	Obsolete and Modified Subroutines	5-13
5.13	#include Files	5-14
6.	Miscellaneous Compatibility Issues	6-1
6.1	Operating System, Communications, and Hardware Issues	6-1
6.2	Porting Device Drivers	6-6
6.2.1	Mapping and Unmapping Devices	6-7
6.2.2	Manipulating the User's Virtual Region	6-8
7.	Troubleshooting Tips	7-1

List of Tables

Table 3-1.	Default IRIS-4D Series Compiler Directives	3-5
Table 3-2.	Unsupported Series 2000 and 3000 Compiler Directives	3-6
Table 5-1.	Comparison of Old and New Subroutines	5-4
Table 5-2.	Subroutines Not Supported by IRIS-4D Series Workstations	5-13
Table 5-3.	Subroutines Supported by Only the Non-GT Workstations	5-13
Table 5-4.	Subroutines that Work Differently	5-14
Table 7-1.	Graphics Tips	7-2
Table 7-2.	Language and Compiler Tips	7-4
Table 7-3.	Performance Tips	7-6
Table 7-4.	Hardware Tips	7-7

List of Figures

Figure 5-1. A Solid Concave Polygon 5-10

1. Introduction

This document describes issues that arise in porting applications written in C or FORTRAN from an IRIS Series 2000 or 3000 workstation to any IRIS-4D Series workstation. By using the information in this document, you can write graphics code that will run on any IRIS-4D Series workstation that is running the 4D1-3.0 software release. It covers these topics:

- C compiler (*cc*) and linker (*ld*) switch support and compatibility
- FORTRAN compiler (*f77*) and linker (*ld*) switch support and compatibility
- interlanguage calling on the IRIS-4D Series workstations
- graphics compatibility
- operating system, networking, and hardware differences; information on porting device drivers
- troubleshooting your port

Chapter 5, “Graphics Compatibility”, gives an overview of differences in graphics. Section 2 of this document, “IRIS 3000 to 4D Conversion Tutorial”, covers several of these differences in more depth through both textual discussion and sample code. Read Chapter 5 first, then see Section 2 for more information.

Note: Because the IRIS Series 2000/3000 workstations use a different processor than the IRIS-4D Series workstations, any application written using assembly or machine language on the former must be rewritten for the latter.

2. Porting C Code

This chapter covers porting code from IRIS Series 2000 and 3000 machines to the IRIS-4D Series workstations. The chapter covers supported and unsupported *cc* and *ld* switches, as well as code differences between the machines.

2.1 Command Line Switches

This chapter contains lists of all supported and unsupported Series 2000 and 3000 compiler and linker switches.

2.1.1 Supported *cc* Switches

The IRIS-4D Series *cc* compiler driver supports the following Series 2000 and 3000 compiler driver switches.

- c** Suppress the loading phase of compilation.
- C** Prevent the macro preprocessor from removing C-style comments.
- E** Run the macro preprocessor and send to standard output.
- Idir** Look in directory *dir* for missing include files. **Caution:** On IRIS-4D Series workstations, **-I** without a *dir* argument will cause the compiler not to look in default directories. This is not supported on Series 2000 and 3000 machines.
- O** Do optimization.

- p** Do profiling.
- P** Run only the macro preprocessor and place results in *file.i*.
- o** Name the final output file.
- Dname** Define *name* to preprocessor.
- Uname** Remove any initial definition of *name*.
- Zg** Load necessary graphics files and libraries.
- S** Leave assembly language in *file.s*.
- g** Generate debugging information.

2.1.2 Supported *ld* Switches

The IRIS-4D Series loader supports these Series 2000 and 3000 loader switches:

- e** The following argument is taken to be the name of the entry point of the loaded program.
- lx** Search the specified library *libx.a*. **Caution:** **-lx** must be placed after the modules containing references to library *x*.
- M** Produce a primitive load map.
- n** Make the text portion read only. **Caution:** When using *f77* to call *ld*, this switch is the default on Series 2000 and 3000 machines, but is not supported as such on IRIS-4D Series workstations.
- o** Name output file.
- r** Generate relocation bits.
- s** “Strip” the output by removing all symbols except locals and globals.
- T** Set text segment origin with next argument.
- u** Take following argument as symbol and enter it as undefined in symbol table.

- x** Do not preserve local symbols in the output symbol table.
Caution: This switch is the default on Series 2000 and 3000 machines, but is not supported as such on IRIS-4D Series workstations.
- ysym** Trace symbol *sym*.
- z** Make the file “demand paged”.
- d** Force definition of common, even if **-r** flag is present.
- S** Strip all output, except for locals and globals.
- v** Output each file as processed.

2.1.3 Unsupported cc Switches

The following IRIS Series 2000 and 3000 switches are not supported by the IRIS-4D Series *cc* compiler:

- ZA** Pass remaining string to *as*. **Note:** Use this switch combination in place of **-ZA**: **-ta -Wa,string**.
- ZC** Pass remaining string to *ccom*.
- Zf** Cause instructions for floating point accelerator to be generated. **Note:** This switch is transparent on the IRIS 4D because it is handled automatically.
- ZF** Pass remaining string to FORTRAN front end.
- Zi** Use the specified file as the run-time startup, rather than the standard C run-time startup. **Note:** Use this switch combination in place of **-Zi**: **-tr -hpath -B**, where *path* is the pathname to the startup files *crt1.o* and *crt0.o*.
- ZN** Pass **-N** switch to *cpp*. **Note:** Use this switch combination in place of **-ZN**: **-Wp,-N**.
- ZP** Pass remaining string to *pc* front end.
- Zz** Print all calls to *exec()*. The IRIS-4D Series equivalent is **-v**.
- x** Keep local symbols in output symbol table and suppress default. This is the default on IRIS-4D Series workstations.

- n** Cause loader to not load program with shared text. This is the default on IRIS-4D Series workstations.
- Zq** Time all subprocesses.
- Zr** Load necessary remote graphics files and libraries.
- Zv** Give additional diagnostics.
- ZRlibroot** Pass **-R** switch to *ld*. The **-L** flag provides a similar function.

2.1.4 Unsupported *ld* Switches

The following Series 2000 and 3000 switches are not supported by the IRIS-4D Series version of *ld*:

- A** Specifies incremental loading; the resulting object may be read into an already executing program.
- D** Pad data segment.
- N** Do not make text portion read only.
- Rlibroot** Use *libroot* for the search path of the files named in the **-l** switch.
- t** Print name of each file as it is processed. The IRIS-4D Series equivalent is **-v**.
- X** Save local symbols, except for label names.

2.2 Code Compatibility

Few changes need be made to Series 2000 and 3000 C code to allow it to run on the IRIS-4D Series *cc* compiler.

2.2.1 Single and Double Precision

The C compiler on Series 2000 and 3000 workstations defines the types **float** and **double** as single precision, and the type **long float** as double precision. On IRIS-4D Series workstations, **double** is supported as double precision. If you need single precision performance, you can use the **-float** option with *cc(1)* to prevent type promotion in expressions, and ANSI C prototypes, as used in the graphics library, to prevent unnecessary **float** to **double** promotion (see */usr/include/gl.h*). For more information, see the *C Language Reference Manual*.

C on the IRIS Series 2000 and 3000 does not convert **floats** to double precision in expressions, or when passing them as arguments in functions. C on the IRIS 4D converts **floats** to double precision in expressions, or when passing them as arguments in functions. C on the IRIS 4D also extends **floats** to double precision (**double**) when passing them as parameters, unless a corresponding function prototype is in force. To avoid this overhead, you must create appropriate prototypes for their functions. (See Chapter 6 of the *C Reference Manual* for information on appropriate prototypes.) This difference affects functions whose floating-point parameters are accessed as non-floating point data, as such functions on IRIS-4D Series workstations must use the *varargs* facility (see section 2.6).

When the overhead associated with promoting **floats** to **double** in expressions is not justified by the added precision, you can specify no promotion by using the **-float** switch with *cc(1)*. This switch informs the compiler that expressions whose highest data type is **float** (i.e., where no **doubles** occur) retain sufficient precision if they are evaluated in single precision. Use of the **-float** switch does *not* affect coercion of **floats** to **doubles** in parameter lists.

Note: Type **double** is synonymous with type **float** (32 bits) on Series 2000 and 3000 systems, but is synonymous with type **long float** (64 bits) on the IRIS-4D Series. Make sure that arguments in format statements for *scanf* and *printf* reflect this change.

2.2.2 Trigonometric Functions

The UNIX library trig functions on Series 2000 and 3000 systems do not conform with System V standards. For performance reasons, the default trig library functions are single precision, not double precision. Double precision trig functions have names prepended with “_l”.

The IRIS-4D Series UNIX library more closely conforms to System V standards, and supports only double precision trig functions. Trig function names with prepended “_l” will compile correctly when *math.h* is included.

If your code uses *math.h* and the code contains redundant declarations for functions that are defined in *math.h*, you receive warnings from the C compiler. This can be alleviated by removing the redundant declarations.

Note: No C code changes are required for programs that include *math.h*; however, if you define C library routines in-line, you must change their C code to accommodate the differences in precision.

2.2.3 Unsigned Characters

The C compiler on Series 2000 and 3000 workstations treats characters as signed by default. On IRIS-4D Series workstations, all characters are treated as unsigned by default. To avoid conflict, use the **—signed** switch on IRIS-4D Series workstations when compiling ported code.

Note: The C compiler on IRIS Series 2000 and 3000 workstations, and IRIS-4D Series systems recognizes the reserved word **signed** as well as **unsigned** (see section 2.7). New code which must be aware of the signed-ness property of characters should specify either **signed** or **unsigned** when declaring variables of this type.

2.2.4 C Data Files

The Series 2000 and 3000 C compiler aligns variables differently from the IRIS-4D Series C compiler. On the IRIS Series 2000 and 3000 workstations, character data is unaligned and all other data is aligned on 2-byte boundaries. The IRIS-4D Series workstation alignment rules are much more restrictive. See Section 2.2.6, “Miscellaneous Code Differences”, for more information. As a result of these alignment differences, you must be careful

when you write code that will write and read data files on both IRIS Series 2000 and 3000 workstations, and IRIS-4D Series workstations. Such code should follow these guidelines:

1. Floating-point values should always be read and written using double precision (i.e., **long float** on the Series 2000 and 3000, **double** or **long float** on the IRIS-4D Series).
2. Structures should be written in such a way so as to force the alignment of elements to be that of IRIS-4D Series workstations. You can force alignment by:
 - placing the element with the most strict alignment as the first element of the structure or union.
 - ensuring that the offset of each subsequent element is correct with its natural alignment. (Future problems will be avoided if the natural alignment of all floating-point data is assumed to be 64-bits.)

2.2.5 The BSD Compatible Library

IRIS-4D Series workstations provide the Berkeley 4.3 compatible library in */usr/lib/libbsd.a*. If you need to include the BSD library, compile your program with this command:

```
cc -I/usr/include/bsd filename -c
```

To link object files together with the BSD library to create an executable file, use this command:

```
cc -o output_file filename.o -lbsd
```

See the *intro(3M)* manual page for more information.

If you are porting from an IRIS Series 3000 workstation running Release 3.4 or an earlier release, substitute the *#include <sys/dir.h>* file for *<ndir>* in your programs on the IRIS 4D.

2.2.6 Miscellaneous Code Differences

The following differences exist between the implementation of C on IRIS-4D Series workstations and that on the Series 2000 and 3000.

Note: Since some Series 2000 and 3000 C programs rely on implementation-defined behavior, you should carefully consider these areas of incompatibility when porting your code.

- **Post-incrementing (decrementing) in parameter lists.** The ANSI C standard specifies that the effect of a postfix auto-increment (decrement) operator on a variable in a parameter list may be delayed until the call is made. This liberty is taken by the C compiler on the IRIS-4D Series, and *not* by the C compiler on older IRIS series. Thus, the code sequence

```
extern int count;
foo(count++, count++);
check(count);
```

produces different values for the parameters to *foo* on the IRIS-4D Series as opposed to earlier systems, although the value of the global variable *count* if it is accessed by *foo* agrees between the systems. Do not rely on the (implementation-defined) value of data which has been so altered within a parameter list.

- **Alignment of global data.** Series 2000 and 3000 C aligns adjacent global data elements on a short boundary. IRIS-4D Series C aligns such data on long boundaries unless the data item is a double or an aggregate whose first element is a double. In this case, the global data is aligned on a double boundary. This difference has no effect on ported code.
- **Alignment of structure and array elements.** Series 2000 and 3000 C aligns non-character structure members and array elements on short boundaries. Structure members and array elements of type **char** are not aligned. IRIS-4D Series C aligns each datum on their 'natural' alignment; shorts are aligned on short boundaries and doubles on double boundaries. This may cause errors when using *unions* in ported code.

Note: The alignment of single-precision floating point data on the IRIS-4D Series may be altered in a future release to match that of double-precision.

- **Return values.** When a function of type **[unsigned] short** or **char** is called, Series 2000 and 3000 C relies on the caller to properly mask the

return value. IRIS-4D Series C expects the function to properly mask the value before returning it. This discrepancy is transparent if a declaration of the function type is visible to the caller. If the type of the function is *not* visible to the caller, the convention used by IRIS-4D Series C produces the correct return value, while that used by Series 2000 and 3000 C does not. The IRIS-4D Series convention repairs some bugs existing in current C programs running on the Series 2000 and 3000 when they are ported. Adverse effects occur only in code which relies on Series 2000 and 3000 conventions.

- **Variable numbers of arguments.** IRIS-4D Series C passes a certain number of parameters in registers. The specific registers used depends on the type of the datum. On Series 2000 and 3000 systems, you can use a pointer to access parameters directly from the parameter area. This is not allowed on the IRIS-4D; on these systems, functions whose argument list is variable (e.g., *printf*) must conform to the following rules in order to access these parameters:
 - The *varargs* convention must be used.
 - parameters must be scalar. (e.g., structures may not be passed in such instances). A pointer to the base of an array, rather than the array itself, is passed, so that arrays qualify as *scalar* parameters.
 - the type of the first parameter must be integral or pointer. (e.g., floating-point types are not allowed.)
- **Direct C calls from FORTRAN.** On the IRIS-4D, the class *fortran* is a no-op.
 - On the IRIS 4D, you must append an underbar to the name of the C function that you can call from FORTRAN.
 - On the IRIS 4D, the argument list for the function should *not* be reversed.
- **Bit fields.** Bit fields on IRIS 2000 and 3000 systems are inherently unsigned. Bit fields on the IRIS 4D have their most significant bit interpreted as a sign bit unless they are declared unsigned. On the IRIS 3000, you see a warning message if a bit field is encountered that is not declared as *unsigned*.

Bit fields in the C language tend to be one of its least portable constructs due to the differences in machine byte- and bit- order. Although both conform to standards, the IRIS Series 2000 and 3000 and the IRIS-4D

series differ in several important ways in the semantics of bit fields and their underlying allocation.

One inherently non-portable use of bit fields is to overlay storage. Using bit fields in this fashion necessitates the programmer knowing how storage for a field is allocated, i.e., how the fields are *packed*. Upon encountering a set of adjacent bit fields, the first bit field is placed in newly-allocated storage aligned on an `int` boundary. (The term *boundary* used in this section refers to the indicated boundary relative to the start of the innermost structure which contains the field.) As indicated previously, structures on the IRIS-4D are always aligned on a four-byte boundary (an eight-byte boundary if the first element is a `double`). Structures on the IRIS Series 2000 and 3000 are aligned on a two-byte boundary.

A C restriction on the allocation of bit field storage dictates that a bit field may not straddle an `int` boundary. Thus, upon encountering a bit field, the width of the bit field is compared to the distance (in bits) to the next `int` boundary. If the new field fits, it is packed adjacent to the previous element. If it does not, storage is padded to that boundary before the field is allocated. Unfortunately, the definition of an `int` boundary is different on the IRIS-4D than on machines of the IRIS Series 2000 and 3000. On the IRIS-4D, an `int` boundary is any address which is a multiple of four bytes relative to the start of the structure, while on the IRIS Series 2000 and 3000, this boundary is any multiple of two bytes.

This latter difference has especially important ramifications for porting code which assumes a knowledge of bit field allocation. For example, the bit field-containing structure

```
struct {  
    unsigned a:14,b:22;  
}
```

will have padding bits inserted between *a* and *b* on all IRIS systems. However, the number of padding bits differs: two padding bits on 2000 and 3000 machines, 18 padding bits on the IRIS-4D.

To port code that both uses bit fields and assumes knowledge of their underlying storage allocation, observe these rules:

- Only use fields of type *unsigned*.

- Upon encountering a field *f*, which follows element *e* in a structure, insert a field to pad to the next four-byte boundary if packing *f* adjacent to *e* would cross a four-byte boundary.

2.2.7 New Reserved Words

The C compiler on the IRIS-4D Series is progressing towards implementation of the emerging ANSI standard for C. To this effect, function prototypes have been implemented, as well as the identifier class **volatile**. This implementation of the new standard adds the following to the list of reserved words in C:

```
const  signed  volatile
```

Code in which these identifiers are used for other purposes must be changed.

2.2.8 Profiling

The C compiler on the IRIS-4D Series supports enhanced profiling capability. Basic profiling is provided by the standard UNIX `-p` switch which must only be used in the link step on the IRIS 4D. This method provides data on per procedure execution times in much the same way as it is provided on older IRIS systems. However, unlike these, it does *not* provide procedure invocation counts.

Further control over profiling may be obtained by using the program *pixie*(1). Use *pixie* on an executable (which has *not* been linked with `-p`) to produce a similar executable in which you have inserted instructions to provide whatever profiling functions you want. (Refer to the manual page on *pixie*(1).)

Neither profiling tool, *prof* or *pixie*, can be used on a program with shared libraries. You must relink your program with the unshared versions of the libraries before profiling it.

For more information, see the *IRIS-4D Series Compiler Guide*, Chapter 2.

2.2.9 Standard Libraries

This section lists the appropriate command line arguments for linking functions found in various libraries on the IRIS 4D. These libraries are identified below by the section number and letter used to identify them in the *IRIS-4D Programmer's Reference Manual*, Volume II.

- (3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the *-lc* option. Declarations for some of these functions may be obtained from *#include* files indicated on the appropriate pages.
- (3G) These functions constitute the IRIS Graphics Library which are documented in the *Graphics Library User's Guide*. If the *-Zg* flag is specified, the C compiler searches this library. Declarations for these functions may be obtained from the *#include* file *<gl.h>*. *<device.h>*, and *<get.h>* define other constants used by the Graphics Library.
- (3M) These functions constitute the Math Library, *libm*. The link editor searches this library in response to the *-lm* option to *ld*(1) or *cc*(1). Declarations for these functions may be obtained from the *#include* file *<math.h>*.
- (3S) These functions constitute the “standard I/O package” (see *stdio* (3S)). These functions are in the library *libc*, already mentioned. Declarations for these functions may be obtained from the *#include* file *<stdio.h>*.
- (3B) Berkeley compatibility routines. This library provides compatible implementations of a limited subset of the functions provided by the Standard C Library in the Berkeley 4.3 Distribution of UNIX. Include files needed for routines in this library are in the tree */usr/include/bsd*. It is recommended that the *-I/usr/include/bsd* compiler control be supplied when compiling programs that call (3B) routines. This library will be searched by the loader when the *-lbsd* flag is supplied.
- (3N) These functions constitute the internet network library. Compiling instructions are the same as for (3B) routines.

- (3R) RPC services built on top of Sun's Remote Procedure Call protocol. To compile and link a program that calls any of these routines, use a compile command of the form:

```
cc -I/usr/include/sun -I/usr/include/bsd prog.c \
-lrpcsvc -lsun -lbsd
```

Note that this library is provided as part of the NFS option package, so it may not be present on all systems.

- (3Y) Yellow Pages routines and RPC support routines. This library contains routines that provide a programmatic interface to SUN's Yellow Pages distributed lookup service. The library also contains Yellow Pages versions of standard routines like *getpwent*(3) that are different in a YP environment. The routines that implement the RPC protocol also reside in this library. To compile and link a program that calls (3Y) routines, use a compile command of the form:

```
cc -I/usr/include/sun -I/usr/include/bsd prog.c \
-lsun -lbsd
```

This is required because routines in the (3Y) library call routines in the (3B) library. Note that the order of the libraries must be as given in order for the references to be resolved. This library is provided as part of the NFS option package, so it may not be present on all systems.

- (3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate man pages.

2.2.10 Compiling Large Programs

The following list of suggestions should help ease the compilation of large C programs on the IRIS 4D.

- Do not try to optimize your code until you are finished porting.
- Use the **-g** option if you will be using the symbolic debugger, *dbx*, or *edge* (*adb* is not supported on the IRIS 4D).
- Use the **-G 0** option for all modules when first compiling large programs. After the code is debugged, link your code using the **-bestGnum** option to determine what the best value for the **-G num** switch is, and recompile

all modules using that value. Failure to follow this suggestion could lead to numerous GP relocation errors from *ld*.

- The optimizer no longer optimizes functions and subroutines whose length exceeds the default limit of 500 basic blocks. If your functions and subroutines exceed this limit, and you compile your program without the optimization switch (**-Olimit *n***), you see the message

```
uopt: Warning: function name: this procedure not optimized
because it exceeds size threshold; to optimize this procedure,
use -Olimit option with value >= number.
```

The optimizer has refused to optimize function *function name*, because its length (*number* basic blocks) exceeds the default **-Olimit** value. To optimize this function, you must add **-Olimit *n*** to your compilation step, where *n* is greater than or equal to *number*.

For example, to change the default limit to 600 basic blocks, set the optimization switch to 600 (or greater): **-Olimit 600**.

Note: The relationship between the time required to optimize a function and its length is not linear.

- To obtain a load map of your program, compile all modules with the **-g** switch, and run *nm* on the resulting executable.
- To resolve naming conflicts between user and system routines, use the **-Wl,-v** switch. It passes the **-v** switch to *ld*, which enumerates each module loaded from the libraries.

2.3 Making Libraries

When making your own libraries on the IRIS-4D, pass each module through *ld*(1) before inserting it in the library. Doing this deletes duplicate symbols and lessens the possibility of overflowing *ld*'s symbol table later. For example, before adding the module *foo* to your library, use these commands:

```
ld -r foo.o
mv a.out foo.o
```

3. Porting FORTRAN Code

This chapter covers porting FORTRAN code from IRIS Series 2000 and 3000 machines to the IRIS-4D Series workstations.

The chapter is divided into two sections; the first lists supported and unsupported *f77* and *ld* command line switches, and the second covers code compatibility between IRIS Series 2000 and 3000 machines and IRIS-4D Series machines.

3.1 Command Line Switches

This chapter contains lists of all supported and unsupported Series 2000 and 3000 compiler and linker switches.

3.1.1 Supported *f77* Switches

The IRIS-4D Series *f77* compiler driver supports the following Series 2000 and 3000 compiler driver switches:

- c** Suppress the loading phase of compilation.
- Idir** Look in directory *dir* for missing include files. **Caution:** On the IRIS-4D Series workstations, **-I** without a *dir* argument will cause the compiler not to look in default directories. This is not supported on Series 2000 and 3000 machines.
- O** Do optimization.
- P** Run only the macro preprocessor and place results in *file.i*.

- g Generate debugging information.
- o Name the final output file.
- D*name* Define *name* to preprocessor.
- U*name* Remove any initial definition of *name*.
- Zg Load necessary graphics files and libraries.
- Zr Load necessary remote graphics files and libraries.
- Zq Time all subprocesses.
- Zv Give additional diagnostics.
- S Leave assembly language in *file.s*.

3.1.2 Unsupported f77 Switches

The following Series 2000 and 3000 switches are not supported by the IRIS-4D Series f77 compiler:

- C Prevent macro preprocessor from removing C style comments.
Caution: On IRIS-4D Series workstations, this switch generates runtime subscript checking.
- ZA Pass remaining string to *as*. **Note:** You can use the following switch combination in place of -ZA: -ta -Wa,*string*
- ZC Pass remaining string to *ccom*.
- Zf Cause instructions for the floating point accelerator to be generated. *ld* will automatically generate code for the accelerator if it exists.
- ZF Pass remaining string to the f77 front end.
- Zi Use the specified file as the run-time startup, rather than the standard C run-time startup. **Note:** You can use the following switch combination in place of -Zi:
-tr -h*path* -B, where *path* is the pathname to the startup file *crt0.o*.
- ZN Pass -N switch to *cxx*. **Note:** The following combination of switches can be used in place of -ZN: -Wp,-N

- ZP** Pass remaining string to *pc* front end.
- Zz** Print all calls to *exec()*. See the IRIS-4D FORTRAN switch **-v**.
- x** Keep local symbols in output symbol table and suppress default. This is the default on IRIS-4D Series workstations.
- n** Cause loader to not load program with shared text. This is the default on IRIS-4D Series workstations.
- E** Run only the macro preprocessor *cpp*. **Caution:** On IRIS-4D Series workstations, this switch passes arguments to EFL.
- L** Produce a FORTRAN listing for each FORTRAN source file.
- ZRlibroot** Pass **-R** switch to *ld*.

3.1.3 Supported *ld* Switches

The IRIS-4D Series loader supports these Series 2000 and 3000 loader switches:

- e** The following argument is taken to be the name of the entry point of the loaded program.
- lx** Search the specified library *libx.a*. **Caution:** **-lx** must be placed after the modules containing references to library *x*.
- M** Produce a primitive load map.
- n** Make the text portion read only. **Caution:** When using *f77* to call *ld*, this switch is the default on Series 2000 and 3000 machines, but is not supported as such on IRIS-4D Series workstations.
- o** Name output file.
- r** Generate relocation bits.
- s** “Strip” the output by removing all symbols except locals and globals.
- T** Set text segment origin with next argument.

- u** Take following argument as symbol and enter it as undefined in symbol table.
- x** Do not preserve local symbols in the output symbol table.
Caution: This switch is the default on Series 2000 and 3000 machines, but is not supported as such on IRIS-4D Series workstations.
- ysym** Trace symbol *sym*.
- z** Make the file “demand paged”.
- d** Force definition of common, even if **-r** flag is present.
- S** Strip all output, except for locals and globals.
- v** Output each file as processed.

3.1.4 Unsupported *ld* Switches

The following Series 2000 and 3000 switches are not supported by the IRIS-4D Series version of *ld*:

- A** Specifies incremental loading; the resulting object may be read into an already executing program.
- D** Pad data segment.
- N** Do not make text portion read only.
- Rlibroot** Use *libroot* for the search path of the files named in the *-l* switch.
- t** Print name of each file as it is processed. The IRIS-4D Series equivalent is **-v**.
- X** Save local symbols, except for label names.

3.2 Code Compatibility

Few changes need be made to Series 2000 and 3000 ANSI standard FORTRAN code to allow it to run on the IRIS-4D Series *f77* compiler. Two routines, GETARG and IARGC differ on the IRIS-4D Series. In addition, a number of Series 2000 and 3000 *f77* compiler directives are no longer supported. This chapter also addresses issues concerning interlanguage calls.

See Section 3.2.7, Compiling Large Programs, for information on porting.

3.2.1 Compiler Directives

The IRIS-4D Series compiler supports these Series 2000 and 3000 compiler directives:

```
$INCLUDE filename
$F66DO
$COL72
```

Table 3-1 provides a list of default directives for the IRIS-4D Series compiler and the warnings the compiler produces when it encounters them.

Directive	Warning
\$NOARGCHECK	IRIS-4D Series default
\$NOTBINARY	IRIS-4D Series default
\$SYSTEM	IRIS-4D Series default [underscore (_) only]
\$CHAREQU	IRIS-4D Series default

Table 3-1. Default IRIS-4D Series Compiler Directives

Table 3-2 provides a list of directives which are not supported by the IRIS-4D Series compiler and the warnings the compiler produces when it encounters them.

Directive	Warning
\$ARGCHECK	not supported
\$BINARY	not supported
\$SYSTEM	not supported [percent (%) only]
\$INT2	not supported
\$LOG2	not supported
\$XREF	not supported
\$SEGMENT	not supported

Table 3-2. Unsupported Series 2000 and 3000 Compiler Directives

The IRIS-4D Series *f77* switch **-i2** performs the same functions as `$INT` and `$LOG`.

Note: The `$INCLUDE` option does not correctly handle line numbers. To avoid problems, use `#include` with the correct syntax to include source files.

3.2.2 Functions and Subroutines

The IRIS-4D Series *f77* compiler supports most Series 2000 and 3000 intrinsic functions and subroutines without special invocation. The command line compiler switch **-ZG** loads the remaining Series 2000 and 3000 compatible library routines, allowing you to compile without source code change when referencing these library routines. You should modify your FORTRAN source code to support these standard interfaces, except when you must support code on Series 2000 and 3000 systems as well as IRIS-4D systems. In this case, you should insert macro preprocessor commands to define the boundary requirements for these intrinsic routines. The code can then be compiled conditionally, according to the machine it is running on.

The only changes which you must make in porting FORTRAN code from Series 2000 and 3000 machines are to the indices of the intrinsic functions GETARG and IARGC:

```
SUBROUTINE GETARG(I,C)
INTEGER*4 I
CHARACTER*(*) C

INTEGER FUNCTION*4 IARGC()
```

In subroutine GETARG, the value of the I'th command line argument is returned in the variable C. On Series 2000 and 3000 systems, the 1st argument is the command name; on IRIS-4D Series workstations, the 0th argument is the command name.

On each system, function IARGC returns the index of the last command line argument as measured by the version of GETARG on that system.

A special compiler switch, **-ZG**, loads the Series 2000 and 3000 compatible GETARG and IARGC objects. If possible, it is better to modify your code to use the IRIS-4D versions.

3.2.3 I/O Compatibility

FORTRAN I/O handling differs between the IRIS-4D Series and Series 2000 and 3000 machines in a few ways; the assignment of preconnected unit numbers, the format of FORTRAN files opened with specifier FORM='UNFORMATTED', FORM='BINARY', or BUFFERED.

Carriage Control Processing

The IRIS 4D1-3.0 release FORTRAN compiler no longer processes carriage control information on *stdout* by default. The 4D1-2.0 release compiler processed the carriage control so that the first character from a *print* or *write* statement was interpreted as a FORTRAN carriage control character. This was changed in the 4D1-3.0 release compiler to be compatible with the IRIS Series 2000 and 3000. If, however, you use the **-vms** switch, carriage control information is processed, and the first character in the *print* statement is interpreted as a FORTRAN carriage control character.

Preconnected Unit Numbers

On IRIS-4D Series workstations, FORTRAN preconnects unit 5 to standard input, unit 6 to standard output, and unit 0 to standard error. On Series 2000 and 3000 machines, FORTRAN connects unit 0 to standard input and output, and unit 1 to standard error.

Unformatted and Binary Files

The utility *uconv*(1) converts a FORTRAN unformatted data file either from Series 2000 and 3000 FORTRAN form to IRIS-4D FORTRAN form, or vice versa. *uconv* allows you to port otherwise non-portable data files opened as `FORM='UNFORMATTED'`.

Series 2000 and 3000 FORTRAN binary files have no record markers, thus a general conversion tool can not be written. You can, however, write a FORTRAN program that reads the binary file and writes to an unformatted file. The FORTRAN programmer must know the format of the file. The format of the file is determined by the sequence of variables written to the file. Each `WRITE` statement defines a record that can be read by an equivalent `READ` statement. The unformatted file can then be converted by *uconv*, and read by the ported FORTRAN program running on the IRIS-4D Series machine. If the ported FORTRAN program does not read the binary file in the same format as it was written, changes to the code may be required to read the records of the new unformatted file.

Note: *uconv* can not convert Series 2000 and 3000 FORTRAN data files opened as either `FORM='BINARY'` or `FORM='UNFORMATTED'` with the `$BINARY` option. Also, IRIS-4D FORTRAN will not give an error message for bad `FORM` specifiers.

Buffered Files

FORTTRAN on the IRIS 4D does not support the `OPEN` statement. All FORTRAN I/O is buffered on the IRIS 4D. IRIS 4D FORTRAN and IRIS Series 3000 FORTRAN are not compatible for mixed C and FORTRAN routines that read and write to *stdin* and *stdout*.

3.2.4 Non ANSI Standard Code

In general, non ANSI standard code is not portable. The IRIS-4D Series *f77* compiler may not interpret non-standard code in exactly the same way as the Series 2000 and 3000 compiler. The following are examples of non-standard code which are interpreted differently by the two compilers:

```
100      DO 100, I = 1, 10
          I = I + 1
          CONTINUE

          CALL XXX (A, B, C)
          .
          .
          .
          ENTRY XXX ()
```

3.2.5 Miscellaneous Code Differences

The following differences exist between the implementation of FORTRAN on IRIS-4D workstations and that on the Series 2000 and 3000.

- The external names generated for FORTRAN subroutines differ between the IRIS-4D and the IRIS Series 2000 and 3000. On the IRIS-4D, the external symbol for a FORTRAN subroutine consists of the subroutine name (truncated to 32 characters), with all characters in lower case, and with an appended underbar. Thus, the external name generated for the FORTRAN subroutine *AnySub* would be *anysub_*.

On the IRIS 2000 and 3000, the external symbol generated for a FORTRAN subroutine consists of the subroutine name (truncated to 31 characters), with all characters in lower case.

- The FORTRAN compiler of the 4D1-3.0 release incorporates enhancements to allow execution of programs which previously failed due to misaligned data references. See the *IRIS 4D1-3.0 FORTRAN Release Notes* or the *FORTRAN Language Reference Manual* for more information.

- The optimizer no longer optimizes functions and subroutines whose length exceeds the default limit of 500 basic blocks. If your functions and subroutines exceed this limit, and you compile your program without the optimization switch (**-Olimit** *n*), you see the message

```
uopt: Warning: function name: this procedure not optimized
because it exceeds size threshold; to optimize this procedure,
use -Olimit option with value >= number.
```

The optimizer has refused to optimize function *function name*, because its length (*number* basic blocks) exceeds the default **-Olimit** value. To optimize this function, you must add **-Olimit** *n* to your compilation step, where *n* is greater than or equal to *number*.

For example, to change the default limit to 600 basic blocks, set the optimization switch to 600 (or greater): **-Olimit** 600.

Note: The relationship between the time required to optimize a function and its length is not linear.

- The 4D1-3.0 compiler contains numerous VMS extensions. See Appendix E in the *FORTRAN Language Reference Manual* for more information.
- The IRIS 4D compiler only supports 19 continuation lines following ANSI standards; The IRIS 2000 and 3000 compiler supports up to 100 lines.
- The compiler on the IRIS-4D does not allow common blocks to have the same names as subroutines.
- To allow the FORTRAN front end on the IRIS-4D Series to understand the non ANSI-standard character \ (backslash), you must change all incidences of the character in the code to read \\ (backslash backslash). This is done to be compatible with C. Be careful not to overflow your code past column 72.
- The 4D1-3.0 compiler conforms to the ANSI standard, when you use the **-static** switch. However, in Release 4D1-3.0, if you use the default mode, (**-automatic**), the compiler does not accept intermixed data and function statements. This will be fixed in a future release.

3.2.6 Compiler Memory-Alignment Extensions

The IRIS-4D RISC architecture imposes certain rules governing how data may be aligned in memory. A variable or array element of size n bytes must be aligned on a boundary whose address is a multiple of n bytes, up to a maximum of eight bytes. Release 4D1-3.0 provides a trap handler and alignment switches to allow access to data which are not so aligned. See version 1.1 of the *FORTRAN Language Reference Manual*.

3.2.7 Compiling Large Programs

The following list of suggestions should help ease the compilation of large FORTRAN programs on the IRIS 4D.

- Do not try to optimize your code until you are finished porting.
- Use the `-g` option if you will be using the symbolic debugger, *dbx*, or *edge* (*adb* is not supported on the IRIS 4D).
- FORTRAN on the IRIS-4D does not initialize all user variables to zero. To work around this problem, follow these steps:
 1. Compile your program with the `-static` switch.
 2. Compile the same program with the `-automatic` switch.
 3. Compare the results of this program with the results of the program with the results from the program compiled with the `-static` switch. Different results between the two programs mean that there are uninitialized variables in your program.
 4. If you get different results, you must debug your program.
- Use the `-G 0` option for all modules when first compiling large programs. After the code is debugged, link your code using the `-bestGnum` option to determine what the best value for the `-G num` switch is, and recompile all modules using that value. Failure to follow this suggestion could lead to numerous GP relocation errors from *ld*.
- When using the optimizer, you must break up programs with more than 12,000 lines of source code into smaller modules. Use `-Olimit N` to override the 500 basic block limit. The optimizer is very slow for functions with more than 500 basic blocks.

- To obtain a load map of your program, compile all modules with the `-g` switch, and run `nm` on the resulting executable.
- To resolve naming conflicts between user and system routines, use the `-WI,-v` switch. It passes the `-v` switch to `ld`, which enumerates each module loaded from the libraries.
- If your processor had subscript range checking, or you suspect that an array reference is out of bounds, use `-C`, which enables runtime array subscript range checking.
- When you compile very large FORTRAN programs (over 20,000 lines) with `-g`, you see this error message:

```
st_fadd: number of files (4095) exceeds max (4095)
```

This internal error occurs because your program exceeds the maximum number of COMMON and EQUIVALENCE statements. This restriction will go away in a future release. The short term solution is to break the file by subroutine into separate files. The system can deal with up to 4095 files.

3.3 Making Libraries

When making your own libraries on the IRIS-4D, pass each module through `ld(1)` before inserting it in the library. Doing this deletes duplicate symbols and lessens the possibility of overflowing `ld`'s symbol table later. For example, before adding the module `foo` to your library, use these commands:

```
ld -r foo.o
mv a.out foo.o
```

4. Interlanguage Calls

Series 2000 and 3000 machines support ‘‘wrappers’’ which rearrange the stack and allow C and FORTRAN to call each other. These wrappers also de-reference FORTRAN parameters to allow C routines to receive by value.

Wrappers are implemented on IRIS-4D Series workstations to call C from FORTRAN only. They are transparent to makefiles already using wrappers. The IRIS-4D Series workstations also support interlanguage calling without wrappers, which is the interface of choice when writing new code. The wrapper generator, *mkf2c(1)*, also has several new features. See the *mkf2c(1)* manual page for further information and Appendix C of the *FORTRAN Language Reference Manual*.

You should modify Series 2000 and 3000 code that does not use wrappers for interlanguage calling, and code that uses wrappers to call FORTRAN from C.

Code that does not use wrappers must adhere to the following IRIS-4D Series calling conventions:

- You must pass arguments on the stack in the same order in both languages.
- You must represent FORTRAN character strings that are parameters as a pointer and a length. Pass the pointer to the string in the usual argument position. Pass the length as an additional parameter appended to the argument list, where the length of the first character string is the first additional parameter, the length of the second character string is the second additional parameter, etc. You must add these additional parameters when calling FORTRAN from C, and can use them to find the end of the string (FORTRAN character strings are not null-terminated) when calling C from FORTRAN.

- C routines may not pass or receive value parameters when interfacing to FORTRAN code. All parameters must be accessed indirectly by using pointers. This restriction does *not* include the special parameters added to the end of the argument list for character string lengths, which are *value* parameters.
- External names of FORTRAN subroutines and functions are generated by the FORTRAN compiler by changing any uppercase characters to lowercase, and by appending an underbar. The subroutine, *Foo* (in FORTRAN source) would have the external name *foo_*, which is visible to C.

5. Graphics Compatibility

This chapter describes the differences between graphics on the IRIS Series 2000 and 3000 workstations and IRIS-4D Series workstations, and how these differences affect porting graphics code. Section 2 of this book provides more detailed information on converting your code.

The Graphics Library for the IRIS-4D Series workstations is basically a superset of the Graphics Library for IRIS Series 2000 and 3000 workstations. It has new subroutines and features that take advantage of the different graphics hardware. However, because the underlying hardware is different between these workstations, some incompatibilities exist. This chapter covers these issues:

- window manager
- screen resolution
- new drawing subroutines
- Gouraud shading
- drawing modes
- overlays and underlays
- cursors
- display modes
- concave polygons
- feedback parsing
- texports and *wsh*
- obsolete and modified subroutines
- *#include* files

5.1 The Window Manager

The IRIS windowing system, 4Sight, runs by default when you log in. Programs that do not use the window manager on the IRIS Series 2000 and 3000 can still run on 4Ds in one of two ways:

- `ginit` opens a full-screen window on the IRIS 4D. When the user presses the right mouse button, no menu appears. `ginit` works just as it did on the IRIS Series 2000 and 3000 workstations.
- `winopen` can open a full-screen window like `ginit`, but this window has a title bar, and will accept user input through the pop-up menus. The drawback to this approach is that `winopen` does not work the same way as `ginit` does with textports; you must use `wsh` instead. See section 5.10, “Textports and `wsh`”.

4Sight replaces *mex* as the IRIS windowing system. 4Sight looks visually different from *mex*, and introduces a few incompatibilities:

- 4Sight does not read the *.mexrc* file at login. Instead it reads *user.ps*.
- 4Sight does not read the *.deskconfig* file at login. You can add the same type of information that was in *.deskconfig* to your *user.ps* file.
- The 4Sight window menu does not have an explicit “attach” item. Input focus automatically goes to the window that is under the cursor. If you want input to go to a window when the cursor is not over it, follow these steps:

1. Position the cursor over the window.
2. Press any key.
3. While pressing the key, move the cursor out of the window.

As long as you keep the key depressed, the input continues to go to that window.

4Sight has a number of new features that are described in *Getting Started with the IRIS-4D Series Workstation* and Section 3 of the *4Sight User's Guide Volume 1*.

For more information on porting code to the window manager, see Section 2 of this book, “IRIS 3000 to 4D Conversion Tutorial”, Chapter 2.

5.2 Screen Resolution

The screen resolution of the IRIS 4D is 67% higher than that of the IRIS 2000 and 3000 Series workstations. The new screen size is 1280 pixels horizontally by 1024 vertically, where the old screen size was 1024 by 768 pixels. This change is reflected in graphics subroutines that take or return absolute screen coordinates as values. Subroutines that accept or return horizontal screen values of 0 to 1023 and vertical screen values of 0 to 767 on IRIS 2000 and 3000 systems can accept or return horizontal screen values of 0 to 1279 and vertical screen values of 0 to 1023 on the IRIS 4D.

The higher screen resolution causes some visual differences:

- cursors, raster fonts, screen images, and icons are about three-quarters the height and width of those on IRIS Series 2000 and 3000 screens
- patterns and line styles are finer
- all user-defined states that use the IRIS 2000 and 3000 screen coordinates look different on the IRIS 4D

To make your code more robust so it can adapt easily to changing screen resolutions, use `XMAXSCREEN` and `YMAXSCREEN` instead of absolute screen coordinates. This lets you specify coordinates as a fraction of the resolution of the screen. For example, instead of specifying the center of the screen with $x = 512$ and $y = 640$, use $x = XMAXSCREEN/2$ and $y = YMAXSCREEN/2$.

For more information, see Section 2, “IRIS 3000 to 4D Conversion Tutorial”, Chapter 3.

5.3 New Drawing Subroutines

Software release 4D1-3.0 introduced several new Graphics Library subroutines for drawing and pixel access. Silicon Graphics recommends converting old style routines to the new ones for three reasons:

- Your code will be more portable.
- On the GT and future products, the new subroutines will run up to 10 times faster than their old counterparts.

- The new subroutines simplify the Graphics Library and allow for future expansion.

In most cases, the conversion is simple — just substitute the new subroutines for the old ones. Unfortunately, the new subroutines do not work in display lists, so if your code is based primarily on display lists, the solution is not so simple.

This table gives a comparison of old and new subroutines.

Technique	Old Subroutines	New Subroutines
draw connected line segments	move, draw, draw	bgnline, v3f, v3f, endl
draw closed hollow polygons	move, draw, draw or poly	bgnclosedline, v3f, v3f, endclosedline
draw filled polygons	pmv, pdr, pdr, pclos polf or splf	bgnpolygon, v3f, v3f, endpolygon
draw points	pnt, pnt	bgnpoint, v3f, v3f, endpoint
read pixels	readpixels, readRGB	rectread, lrectread
write pixels	writepixels, writeRGB	rectwrite, lrectwrite
draw triangular meshes	new	bgntmesh, v3f, v3f, endtmesh
color(vector)	RGBcolor	cpack or c3i
surface normal	normal	n3f
clear screen, Z-buffer	clear, zclear	czclear
create RGB writemask	RGBwritemask	wmpack

Table 5-1. Comparison of Old and New Subroutines

For more information on these subroutines, see the *GT Graphics Library User's Guide*, and *Tuning Graphics Code for Your IRIS-4D Workstation*.

5.4 Gouraud Shading

Gouraud shading is a mode for drawing polygons that is toggled by `shademodel`. The default mode is `GOURAUD`; this slows performance considerably on the non-GT workstations, but makes no difference on the GT. You should use this mode on the non-GT workstations only when Gouraud shading is necessary. When your program is not drawing Gouraud-shaded polygons, the `shademodel` subroutine should have this form:

```
shademodel (FLAT) ;
```

For more information, see Section 2, “IRIS 3000 to 4D Conversion Tutorial”, Chapter 4.

5.5 Drawing Modes

The IRIS 4D uses the concept of drawing modes to shift between the 24 bitplanes used for normal displays, and the special bitplanes used for overlays, underlays, pop-up menus, and cursors. Drawing modes are changed using the `drawmode` subroutine:

```
drawmode (mode)  
long mode;
```

On all IRIS-4D Series workstations, `drawmode` supports five modes:

- `NORMALDRAW`, which sets operations for RGB and color map modes;
- `OVERDRAW`, which sets operations for the overlay bitplanes;
- `UNDERDRAW`, which sets operations for the underlay bitplanes;
- `PUPDRAW`, which sets operations for pop-up menus;
- `CURSORDRAW`, which sets operations for the cursor bitplanes.

Some IRIS-4D Series workstations support additional modes. For more information on `drawmode`, see Section 6.8 of the *Graphics Library User's Guide*, Volume I, that was shipped with your IRIS-4D. See also Section 2 of this book, “IRIS 3000 to 4D Conversion Tutorial”, Chapter 5.

5.6 Overlays and Underlays

On the IRIS Series 2000 and 3000, static overlays and underlays require the use of `writemask`, which reduces the number of bitplanes available for other purposes. The allocation of each bitplane halves the number of screen colors available for other uses. Also, overlays and underlays may not be used in RGB mode on an IRIS 2000 or 3000.

On the IRIS 4D, two bitplanes are reserved for static overlays or underlays. These bitplanes are not shared with the standard bitplanes, so no colors are lost. Overlays and underlays are compatible with RGB mode on IRIS-4D Series workstations.

To initialize the bitplanes for overlays, use this code sequence once:

```
overlay (numplanes) ;  
gconfig() ;
```

numplanes is the number of bitplanes used, and should be either 0 or 2.

Once the bitplanes are initialized, set the current drawing mode to `OVERDRAW`.

```
drawmode (OVERDRAW) ;
```

Then you can define colors, set a writemask, and draw an object in the bitplane(s).

To initialize the bitplanes for underlays, use this code sequence once:

```
underlay (numplanes) ;  
gconfig() ;
```

numplanes is the number of bitplanes used, and should be either 0 or 2.

Once the bitplanes are initialized, set the current drawing mode to `UNDERDRAW`.

```
drawmode (UNDERDRAW) ;
```

Then you can define colors, set a writemask, and draw an object in the bitplane(s).

Note: Overlays and underlays cannot be used simultaneously.

For more information, see Section 2, “IRIS 3000 to 4D Conversion Tutorial”, Chapter 5.

5.7 Cursors

On the IRIS Series 2000 and 3000, the cursor icon is written directly into the standard bitplanes. The IRIS 2000 and 3000 systems allow a cursor to be only one size (16x16).

On the IRIS 4D, the cursor is not stored in any of the bitplanes. The drawback to this is that there is now no way to guarantee that your cursor is a different color from the background. The IRIS 4D system also provides new cursor capabilities. A cursor can be defined as either a 16x16 or 32x32 pixel pattern or as a full-screen cross-hair. Pattern cursors can consist of up to three colors (two bits).

The IRIS Series 2000 and 3000 subroutine `defcursor` is supported by the IRIS 4D, but the cursor type must be declared before the subroutine is called. The valid cursor types for all current and future 4Ds are `C16X1`, `C32X1`, and `CCROSS`. All current 4Ds support a 2-bit cursor, so `C16x2` and `C32x2` are also valid; however, to keep your code as portable as possible for future 4Ds, Silicon Graphics recommends using only the first three cursor types.

Use the following procedures to create cursors on the IRIS 4D:

```
curstype(type) ;
defcursor(1,bitpattern) ;
curorigin(1,horiz,vert) ;
drawmode(CURSORDRAW) ;
mapcolor(colindex,red,green,blue) ;
drawmode(NORMALDRAW) ;
setcursor(1) ;
```

type is one of the cursor types enumerated above.

bitpattern is an array which defines the bit pattern for the cursor. For single-color cursors, this array would consist of 16 values of type `short` for a 16x16 cursor, and 32 values of type `long` for a 32x32 cursor.

For multicolored cursors of both sizes, the number of values in each array is doubled. The first half of these values define the pattern for cursor color bit 1 and the second half of these values define the pattern for cursor color bit 2. For both sizes of multicolored cursor, the color index for each pixel is determined from the sum of cursor bits 1 and 2; if a cursor pixel has both of the values turned off, the cursor is transparent at that pixel.

For the full screen cross-hair, *bitpattern* must be a (dummy) null array of type **short**.

horiz and *vert* locate the “hot spot” of the cursor.

colindex is the color index for determining cursor color at a point in the bit pattern of the cursor. For single-color cursors, this value should be 1. For multicolored cursors, make two additional calls to `mapcolor`, using color indices of 2 and 3, respectively.

red, *green*, and *blue* are integers between 0 and 255 that collectively determine the color of the given index.

`setcursor` and `RGBcursor` do not use colors or writemasks on the IRIS 4D, because the cursor is not written into the standard bitplanes, and cannot be writemasked with the rest of the colors. The cursor always appears with the same colors no matter what is underneath it. Giving an IRIS 4D cursor multiple colors helps to prevent it from "disappearing" over background colors.

Cursor colors on the IRIS 4D aren't fixed; you can change them in your programs. However, changing any color index changes all 4Sight cursors.

For more information, see Section 2, “IRIS 3000 to 4D Conversion Tutorial”, Chapter 6.

5.8 Display Modes

Unlike the IRIS 2000 and 3000 Series workstations, IRIS-4D Series workstations support at least 24-bit RGB color in single-buffer mode and 12-bit RGB color in double-buffer mode. In addition to this, the window manager supports multiple windows running different color display modes (RGB or color map) simultaneously.

Because RGB mode and double-buffer mode are not mutually exclusive on IRIS-4D Series workstations, certain code sequences may cause different effects. For example, consider this sequence:

```
doublebuffer();  
gconfig();  
RGBmode();  
gconfig();
```

On the IRIS Series 2000 and 3000, this puts you into single-buffered RGB mode. On the IRIS-4D Series, this puts you into double-buffered RGB mode.

Note: All open windows on the IRIS 4D that use the color map must share it as they do on the IRIS 2000 and 3000 Series workstations.

Color map mode, unlike the RGB mode, uses only 12 bitplanes to produce a color map. In addition, the top 256 colors (3840-4095) of the color map are occupied by the gamma ramp by IRIS non-GT workstations; this does not happen on the GT. This may interfere with layering schemes using bitplane partitions.

24-bit Z-buffering for hidden surface removal can be done in all graphics modes at real-time speed, including double-buffer color map and RGB modes. Note that using the Z-buffer does not reduce the number of drawing colors available to you; the Z-buffer hardware is separate from the bitplanes.

The IRIS-4D lighting model facility automatically calculates color using specified properties of defined objects and lights as input. Code that uses the Graphics Library lighting subroutines will run much faster on the GT. For information on how to use lighting models, see Chapter 14 of the *Graphics Library User's Guide*, or the *GT Graphics Library User's Guide*.

For more information on display modes, see Section 2, "IRIS 3000 to 4D Conversion Tutorial", Chapters 5 and 7.

5.9 Concave Polygons

The IRIS 4D supports rendering of concave polygons. On the IRIS Series 2000 and 3000 concave polygons are not properly filled. The polygon fill method works reliably for convex polygons. If you wanted to fill concave polygons on the IRIS Series 2000 and 3000, you needed to split the polygons with your own application program.

On the IRIS-4D, the graphics manager (in the rendering subsystem) separate polygons into trapezoids. By default, the graphics manager is optimized to process only convex polygons. The graphics manager can be set into a different mode which splits concave polygons into trapezoids. In this mode, convex polygons are filled more slowly.

The Graphics Library routine, `concave()`, switches the polygon fill mode.

- `concave (TRUE)` use only if concave polygons may be drawn
- `concave (FALSE)` is the default, optimized for convex polygons

The sample code below draws the concave polygon shown in Figure 9-1.

```
concave (TRUE);  
color (RED);  
pmv2i (100, 100);  
pdr2i (300, 500);  
pdr2i (500, 200);  
pdr2i (x, y);  
pclos ();  
concave (FALSE);
```

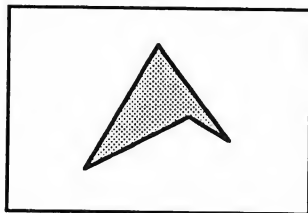


Figure 5-1. A Solid Concave Polygon

When you draw concave polygons, you may encounter one of these problems:

- clipped concave polygons may be filled incorrectly
- Gouraud shaded, concave polygons may be drawn with incorrect colors

5.10 Feedback Parsing

You can use feedback from the Geometry Pipeline to perform matrix multiplication, scaling, and clipping on the IRIS 2000 and 3000 Series and the IRIS-4D Series workstations. However, the data that the feedback subroutines return is highly dependent on the underlying hardware, so your application will need to parse feedback differently for each IRIS-4D Series workstation. Because of this incompatibility, Silicon Graphics discourages the use of these feedback subroutines:

```
feedback
endfeedback
passthrough
xfpt
```

`xfpt`, the feedback subroutine that transforms floating point values, is supported by only the non-GT workstations. It is not supported by the GT, and will not be supported by any future IRIS-4D Series workstations. The sample code below shows how to convert code that uses `xfpt` to completely portable code that can run on any current or future IRIS-4D.

The `xfpt` subroutines multiplied their arguments by the top matrix on the stack, and left the result in the feedback buffer. To port this code, replace the `feedback` call by a `getmatrix` call, and convert each `xfpt` subroutine to a call that does a matrix multiplication of the argument by the matrix returned by `getmatrix`. The resulting values should then be written to the appropriate data structures (wherever the subroutines that parsed the raw feedback buffer would have put them). You should also remove the `endfeedback` subroutine. This conversion should both speed up the code, and make it completely portable.

```

/* this code transforms the point (x, y, z) by the matrix
 * at the top of the stack, and returns the value in
 * xformedpoint.
 */

float x, y, z;
short feedbuf[20];
union {
    float floatdata;
    short shortdata[2];
} alignhack; /* for alignment problem */
long i;
float xformedpoint[4];
feedback(20, feedbuf);
xfpt(x, y, z);
endfeedback(feedbuf);
for (i = 0; i < 4; i++) {
    alignhack.shortdata[0] = feedbuf[1+i*2];
    alignhack.shortdata[1] = feedbuf[2+i*2];
    xformedpoint[i] = alignhack.floatdata;
}

```

It can be replaced by:

```

float mat[4][4];
float xformedpoint[4];
long i;

getmatrix(mat);
for (i = 0; i < 4; i++)
    xformedpoint[i] =
        x*mat[i][0]+y*mat[i][1]+z*mat[i][2]+mat[i][3];

```

5.11 Textports and *wsh*

IRIS-4D Series workstations support textports through a user program, *wsh*, while the IRIS Series 2000 and 3000 workstations support them in the kernel. The textport subroutines in the Graphics Library control *wsh* through a sequence of control characters. If your code uses *winopen* to open a graphics window, the window manager treats *wsh* just as it does any other user program; this means you may have to manually direct input focus to the textport. You can do this by placing the cursor over the textport.

Note: When you reattach to the text window, you disconnect from the graphics window.

5.12 Obsolete and Modified Subroutines

A number of subroutines from the IRIS 2000 and 3000 Series Graphics Library are considered obsolete on IRIS-4D Series workstations. These subroutines do not generate errors on current IRIS 4D products, but serve no useful function. Also, the non-GT workstations support a few subroutines that the GT and all future products will not support. The tables below list both types of subroutines.

Obsolete Subroutines	
callfunc	resetls
devport	RGBcursor
getlsbackup	setshade
getshade	spclos
gflush	textwritemask
gRGBcursor	getmem
lsbackup	winat
pagewritemask	winattach

Table 5-2. Subroutines Not Supported by IRIS-4D Series Workstations

Non-GT Subroutines	
clearhitcode	resetls
gethitcode	RGBcursor
getlsbackup	xfpt
getresetls	setslowcom
gRGBcursor	setfastcom
lsbackup	

Table 5-3. Subroutines Supported by Only the Non-GT Workstations

This table lists which IRIS 2000 and 3000 subroutines have been modified on the IRIS 4D, and how they are different.

Subroutine	Difference
getcursor	some variables return no values
setcursor	
setvaluator	
select	this has been replaced by gselect
defpattern	the 4D does not support 64 x 64 bit patterns

Table 5-4. Subroutines that Work Differently

5.13 *#include* Files

Graphics Library include files reside in the directory */usr/include/gl* on the IRIS 4D.

6. Miscellaneous Compatibility Issues

This chapter covers issues of compatibility between IRIS-4D Series workstations and IRIS Series 2000 and 3000 operating systems, communications (NFS), and hardware. It also shows you how to port device drivers to the IRIS-4D family.

6.1 Operating System, Communications, and Hardware Issues

This sections describes issues that arise when you port your code to the IRIS-4D family from an IRIS Series 2000 or 3000 workstation.

Operating System

- On the IRIS-4D Series, *malloc(3)* is derived from System V, and is different from IRIS Series 2000 and 3000 *mallocs*. On the IRIS-4D Series, the *malloc()* in *libc* allows you to allocate a zero-length block, but *libmalloc*'s *malloc()* function returns NULL on a request to allocate zero bytes. *libmalloc* has better virtual memory properties than the *libc malloc*. For graphics applications, *libmalloc* usually provides much better performance. However, unlike the *libc malloc*, *libmalloc* doesn't allow you to free memory and call *realloc* with a pointer to that memory.
- The *dbx* delete command on the IRIS-4D Series uses commas as the delimiter for deleting multiple events. *dbx* on the IRIS Series 2000 and 3000 uses blanks as the delimiter.

- The `-s` option to the `grep` command generates output on the IRIS-4D Series, but not on the IRIS Series 2000 and 3000, where it returns only the exit codes to the calling program. On the IRIS-4D Series, the `-s` option causes `grep` not to complain if files don't exist.
- The `du` command reports block usage in 512-byte blocks on the IRIS-4D Series and in 1K units on the IRIS Series 2000 and 3000. The `df` command reports block usage in 1K-byte blocks on both the IRIS-4D Series and IRIS Series 2000 and 3000.
- If you log in to an IRIS Series 1000, 2000, or 3000 workstation from an IRIS-4D Series, you may get the message:

```
terminal type iris-ansi-net is unknown
```

To prevent this problem, follow this procedure:

1. Log in as root on the IRIS Series 1000, 2000, or 3000.
2. Edit `/etc/termcap`. Type the lines below at the end of `/etc/termcap`. You must use the tab key for all of the indented lines. Type the up-character (^) just as it appears by pressing the shift key and the 6-key simultaneously. Also, look closely at the text below to distinguish the letter el (l) from the number one (1).

```
# Silicon Graphics IRIS wsh terminal emulator. Can't use vt100
# entry directly because it contains delays. For ease of
# comparison, keep in same order as iris-ansi terminfo entry.
```

```
S8|iris-ansi|IRIS emulating a 40 line ANSI terminal (vt100):P
      :do=^J:sf=ED:co#80:li#40:P
      :cl=E[HE[2J:am:cm=E[%i%2;%2H:P
      :nd=E[C:up=E[A:ce=E[K:cd=E[J:P
      :so=E[7m:se=E[m:us=E[4m:ue=E[m:P
      :ks=E[?1hE=:ke=E[?1lE>:P
      :ho=E[H:sr=EM:P
      :ku=EOA:kd=EOB:kr=EOC:kl=EOD:kb=^H:P
      :k1=EOp:k2=EOq:k3=EOr:k4=EOs:P
      :al=E[L:dl=E[M:P
      :is=E[?1lE>E[?7h:P
      :vs=E[10/yE[=1hE[=2lE[=6h:ve=E[9/yE[12/yE[=6l:P
      :bs:pt:
```

```
# 24-line version
SD|iris-ansi-24:li#24:tc=iris-ansi:
# 66-line version
SE|iris-ansi-66:li#66:tc=iris-ansi:
```

```
# Special value for $TERM so that local and remote shells can be
# distinguished. The network programs do the coercing
# of iris-ansi into this.
S9|iris-ansi-net|IRIS connected to a remote host:tc=iris-ansi:
# 24-line version
SF|iris-ansi-24-net:li#24:tc=iris-ansi-net:
# 66-line version
SG|iris-ansi-66-net:li#66:tc=iris-ansi-net:
```

- Some files that were in */usr/bin* and */usr/include* have been moved. If you cannot find a file in these directories, look in the directories under */usr*, especially */usr/sbin*, */usr/lbin*, and */usr/bsd*. You can put these directories in your path list. A good way to set your path is this:

```
:/usr/bsd:/bin:/usr/bin:/usr/sbin:/usr/lbin
```

- There are two versions of *rsh*: the AT&T restricted shell in */bin* and Berkeley's remote shell in */usr/bsd*. For most users, it is a good idea to put */usr/bsd* early in their path so that they pick up the remote shell.
- *ranlib* is not supported on IRIS-4D Series workstations. It is not required for 4D archives.

- The `-n` option to *echo* does not exist in the Bourne shell on the IRIS 4D. Use ‘`\c`’ instead.
- *ls*, when performed by root on IRIS 2000 and 3000 systems, uses a modified form of the `-a` option. This is not supported on the IRIS-4D. By default, `-C` (column output) is not on.
- The *curses* package, which allows you to write screen management programs, is upwardly compatible from the IRIS Series 2000 and 3000 to the IRIS-4D Series. *terminfo*(4), the IRIS-4D database containing descriptions of terminal capabilities, is a superset of *termcap*(4), the IRIS Series 2000 and 3000 equivalent. Use the *captoinfo*(1M) command to convert *termcap* to *terminfo* descriptions.

You must now use the `-lcurses` flag to compile programs that you formerly compiled with `-ltermcap` or `-ltermLib`. Old makefiles still work because Silicon Graphics currently provides the link to `-lcurses`. For future portability, it is a good idea to change your makefiles.

curses programs are compatible from the IRIS Series 2000 and 3000 to the IRIS-4D Series. IRIS-4D Series *curses*(3X) is a superset of Series 2000 and 3000 *curses*, and is much more powerful than the older version. For example, IRIS-4D Series *curses* supports text windows and multiple terminals. For more information on the *curses/terminfo* package, see ‘*curses/terminfo*’ in the *IRIS-4D Programmer's Guide*.

- On the IRIS-4D Series, a new command, *sar*, gives information about the system performance. This command covers the functionality of the IRIS Series 2000 and 3000 commands *vmstat* and *uptime*. Here is a description of some of *sar*'s most useful options:

The `-u` option gives information about CPU activity, including idle time and kernel activity.

The `-y` option monitors the ttys. This option is useful for debugging programs that make use of serial ports.

sar -m provides about the same information as *vmstat*. The `-w` option, which tells how many context switches are occurring, is useful in combination with the `-m` option. The `-c` option gives the number of system calls.

The *sar -q* provides similar information to *uptime*. Without an additional argument, the sampling rate is coarser. For finer granularity, see the *sar* man page.

The `-r` option reports the free memory pool, in a more comprehensible format than the information *vmstat* provides.

See the *sar* man page for more information.

- Start-up scripts are organized differently. They are now located in */etc/init.d*, and are no longer prefixed with 'rc'. For example, the *tcp* startup script is */etc/init.d/tcp*.
- The *cron* subsystem has changed considerably. *cron* scripts now reside in the directory */usr/spool/cron/crontabs*, with a script for each user id that requires *cron* jobs. A new command, *crontab(1)*, manipulates the scripts.
- The on-line man pages are no longer *troff* source, but rather pre-roffed versions that reside in */usr/catman/*_man/cat[1-7]*. The *man(1)* command no longer processes raw manual pages.
- *varargs(4)* are handled differently on the IRIS-4D.
- The math library *libm.a* differs significantly between IRIS-4D Series and IRIS Series 3000. Most importantly, functions like *sqrt(3M)* now return a *double* rather than a *float*. These changes should be transparent to code that uses *math.h*.
- Certain *mextools* like *mousewarp(1)* are not shipped with the IRIS-4D Series workstation.
- *ps* has different options; for example, to get a listing of all processes with their arguments, use *ps -ef*.
- Device names for disks and tapes are different. In particular, */dev/tape* is always linked to the local streaming tape. */dev/nrtape* is the no-rewind version. Disk device names are now located in subdirectories: */dev/dsk* for block devices, and */dev/rdisk* for character devices.

Networking

- The IRIS-4D NFS option is shipped with a *crontab* file that automatically translates YP maps from the master server to the slave server. To automatically update YP maps, edit the file */usr/spool/cron/crontabs/root* to remove the comment marks (#) from the beginning of the last three lines in the file. See *cron(1M)*.

- *uucp* operates differently on the IRIS-4D Series than on the IRIS Series 2000 and 3000. See the chapter entitled “Basic Networking” in the *IRIS-4D System Administrator's Guide*.
- When using *rsh*, *rcp*, or *rlogin*, you must specify the user and host as *user@host* rather than *host.user*.

Hardware

- The IRIS-4D Series keyboard emulates the IBM PC/RT keyboard; the keyboard of the IRIS Series 2000 and 3000 emulates the VT100. Remote VMS users cannot use EDT, which relies on the keyboard to generate VT100 escape sequences.
- IRIS-4D Series serial ports take 9-pin connectors, while IRIS Series 3000 serial ports take 25-pin connectors.

6.2 Porting Device Drivers

The 4D1-3.0 software release includes new driver-kernel interface routines that allow you to map I/O devices directly into a user's virtual address space. These routines isolate the user-I/O driver from the workings of the virtual memory system, which can change from release to release. If you've written a device driver that manipulates virtual memory, you need to change the driver to use the new interface routines. These changes will make your driver work with release 4D1-3.0 and future IRIS-4D software releases.

With the new routines, you can map either the registers of a device or the contents of a device (for example, a raw disk partition). You can also provide a fast channel between the I/O device interrupt function and the user process, so that most interrupt processing can be done at the user level.

6.2.1 Mapping and Unmapping Devices

The *mmap*(2) system call, new with release 4D1-3.0, provides a user-level interface for manipulating virtual address space. *mmap* maps pages of memory and returns a pointer into the user's address space.

mmap calls the appropriate entry in the block or character device switch table when:

- a user calls *mmap*, and
- the *fd* argument to *mmap* is associated with a special file. For example:

```
fd = open("/dev/special", O_RDWR);
addr = mmap(0, len, prot, flags, fd, off);
```

The entries in the block or character device switch table perform driver-specific mapping and unmapping, and have the form *drvmap* and *drvunmap*, where *drv* is the driver prefix. *mmap* calls *drvmap*; *munmap* calls *drvunmap*. These routines, like all other driver entry points, do not have to be present. If *drvmap* is not present, *mmap* doesn't map the device, and sets *errno* to *ENODEV*.

The arguments to *drvmap* are shown below. The kernel passes these arguments to the driver.

```
drvmap(dev, vt, off, len, prot)
dev_t dev;           /* device number */
vhandle_t *vt;       /* handle to caller's virtual address space */
off_t off;           /* offset into device */
int len;             /* # of bytes to map */
int prot;            /* protections */
```

The *vt* argument is an opaque handle to the virtual space in the calling process where the device will be mapped. This argument represents a data structure within the kernel. The format of this data structure can change from release to release. For compatibility, you can pass the *vt* handle from the *drvmap* routine to other routines. The section below tells you how to access parts of the data structure by using the *vt* handle.

The *off* argument to *drvmap* is the offset in bytes to the beginning of the mapping. This field is device-specific. By convention, it should be the offset (if any) from the beginning of the base address of the device. It could also be an offset into the entire VME bus.

len is the length in bytes of the region that the user wants to map. *prot* is the protection argument from the *mmap()* call.

The arguments to *drvunmap* are shown below.

```
drvunmap(dev, vt)
dev_t dev;      /* device number      */
vhandle_t *vt;  /* handle to caller's virtual address space */
```

The *vt* argument is the same as for *drvmap*, above. The system calls *drvunmap* when you unmap the region, either by calling *munmap* or during *exit*.

munmap() does sanity checking before it calls *drvunmap*. If the sanity checks succeed, *munmap* unmaps the associated virtual memory pages, even if *drvunmap* fails.

Note: If a driver provides *drvmap* but not *drvunmap*, *munmap()* returns *ENODEV*. It is a good idea to provide a dummy unmapping routine if the driver doesn't need to perform any action when it unmaps the device.

6.2.2 Manipulating the User's Virtual Region

The kernel routines described in this section allow you to manipulate the user's virtual region. This section is divided into four parts:

- Allocating memory to share between the driver and an application
- Mapping the contents of a device
- Mapping device registers into the user's address space
- Returning elements of the *vt* argument (opaque handle to the user's address space)

Note: To use the routines described below, make sure to include the files *sys/types.h* and *sys/region.h*.

Allocating Memory to Share

To allocate memory to share between the driver and the application process, the driver map routine follows these steps:

1. Use the *kvpalloc* routine to allocate some pages in the kernel. Use the *btoc()* macro in *sys/sysmacros.h* to convert the *len* argument, which is in bytes, to pages.

```
caddr_t
kvpalloc(btoc(len), flags)
int npages;
int nosleep;
```

kvpalloc allocates physical memory and returns a kernel virtual address associated with that memory. The physical memory is not subject to paging. If the flags argument has the VM_NOSLEEP bit set (this bit is defined in *sys/immu.h*), *kvpalloc* returns NULL if there is not enough available memory to honor the request. Otherwise, *kvpalloc* sleeps until enough memory becomes available.

2. Map the pages into the user's address space by using *v_mapphys*. The address argument, *addr*, is the virtual address returned by *kvpalloc*.
3. To free the memory, use *kvpfree*.

```
kvpfree(addr, npages)
caddr_t addr;
int npages;
```

Mapping the Contents of a Device

To map the contents of a device, the driver map routine calls the *v_mapreg* routine.

```
int
v_mapreg(vt, off, len)
vhandle_t *vt;
int off;
int len;
```

This routine maps the file associated with *vt*, the opaque handle to the user's virtual address space. The mapping begins at offset *off* for *len* bytes into the

user's virtual address space associated with *vt*. The kernel passes these three arguments to the *drvmap* routine. Pages are faulted in on demand, and written to the device in these three cases:

- The user issues an *msync()* command.
- The virtual memory system chooses to reassign the page.
- The user unmaps the address space.

Mapping Device Registers

This section describes two ways to map device registers into the user's address space. The first way is to use the *v_mapphys* routine. The second way is for drivers that only want to map device registers, with no storage involved. In this case, you don't need to write a driver specifically for this purpose; instead, use a general interface via */dev/mmem*, a special file associated with the general memory-mapping driver.

The *v_mapphys* routine sets up a virtual to physical mapping from *vt* to *addr*. *vt* is the opaque handle to the user's virtual address space (see previous section).

```
int
v_mapphys(vt, addr, len)
vhandle_t *vt;
caddr_t addr;
int len;
```

addr can be a physical I/O device address or a kernel address, but it cannot be a user virtual address. Use the *k1seg* address of a physical device, since *k1seg* addresses ensure that the register data are never cached. The address must be page-aligned; the kernel supplies protections only on a page basis. *v_mapphys* returns 0 on success; it sets *errno* and returns -1 on failure.

Caution: Be very careful when you map device registers to a user process. Carefully check the range of addresses that the user requests to make sure that the request references only the requested device. Since protection is available only to a page boundary, configure the addresses of I/O cards so that they

don't overlap a page. If they are allowed to overlap, an application process may be able to access more than one device, possibly a system device (for example, the disk or Ethernet). This is likely to cause problems.

The second way to map device registers is to use the general memory-mapping driver. Follow these steps:

1. Become the superuser.
2. Edit the file `/usr/sysgen/master.d/mem`. Add an entry in the array for *len* and *off*, where *len* is the size in bytes and *off* is the starting page-aligned address of the registers to be mapped.
3. Use *lboot* to reconfigure the set of mappable addresses. See the *lboot* man page and "Configuring a Kernel" in the *IRIS-4D Series Owner's Guide*.
4. When the user calls *mmap*, she or he passes a file descriptor associated with `/dev/mmem`, the special file associated with the general memory-mapping driver.

```
fd = open("/dev/mmem", O_RDWR);  
addr = mmap(0, len, prot, flags, fd, off);
```

len and *off* are the entries that you made in the array in `/usr/sysgen/master.d/mem`.

Returning Data Associated with the Opaque Handle

To return the unique identifier associated with *vt*, the opaque handle to the user's virtual address space, use the *v_gethandle* command.

```
unsigned  
v_gethandle(vt)  
vhandle_t *vt;
```

Since the virtual handle points into the kernel stack, it is likely to be overridden. Use *v_gethandle* if your driver must "remember" several virtual handles.

To return the virtual address in the process where the space is attached, use the *v_getaddr* command.

```
caddr_t  
v_getaddr(vt)  
vhandle_t *vt;
```

To return the length in bytes of the virtual space, sue the *v_getlen* command.

```
int  
v_getlen(vt)  
vhandle_t *vt;
```

7. Troubleshooting Tips

This section lists the most frequent problems you may encounter when you port code from the IRIS Series 2000 or 3000 workstation to an IRIS-4D Series workstation. The problems are divided into four categories:

- graphics
- programming problems and error messages
- performance/speed problems
- hardware

Each section lists the unexpected/undesireable behaviors that you may see, tells you what may be causing the behaviors, and recommends ways to fix or work around them.

Graphics

Problem	Cause	Solution
Colors are wrong or non-existent.	If your program runs in RGB mode, the gamma correction hardware uses the top 256 colors.	When using RGB mode, do not use the top 256 (3840-4095) colors.
	If you are using depth-cueing and the colors are wrong, the near and far clipping planes may also be reversed. This is because the arguments to <code>setdepth</code> are different.	See the <code>setdepth</code> man page in the <i>Graphics Library User's Guide</i> .
No graphics appear in several windows at once.	This happens when the program uses multiple windows per process, and runs in more than one graphics mode. Windows are swapping buffers at different times.	Call <code>swapbuffers</code> for each individual window.
The window manager won't read input.	If the program calls <code>ginit</code> , the system displays one window that takes over the screen, and attaches all input to the window. The window manager then ignores requests for pop-up menus. This provides an environment that is compatible with <i>mex</i> on the 2000 and 3000. runs on the 4D.	Use <code>winopen</code> instead of <code>ginit</code> . This makes the program accept requests for pop-ups, but your textports work differently, and <code>tpoff</code> will not work.

Window positions are wrong, characters are small, software picking doesn't work correctly.

The new screen has higher resolution (1280x1024). Any positions you set using absolute screen coordinates will look different.

Don't specify absolute screen positions ($x=640$, $y=512$). Specify positions as fractions of XMAXSCREEN and YMAXSCREEN ($x=XMAXSCREEN/2$, $y=YMAXSCREEN/2$). This way positions will be correct regardless of screen resolution.

The program crashes, the Graphics Library dumps core.

This will happen if you assign your own subroutines the same name as Graphics Library subroutines.

Make sure your own subroutine names do not conflict with Graphics Library subroutines. Some very common conflicts are normal and rotate.

Windows are not redrawn.

This happens if you use PIECECHANGE tokens. These tokens are no longer supported.

Use REDRAW tokens instead.

Table 7-1. Graphics Tips

Languages and Compilers

Problem	Cause	Solution
Bus error followed by a core dump. (Note: this can be caused by several different problems — this is the most common cause.)	IRIS-4D Series workstations require data alignment. Floating point numbers have a 4-byte boundary; double precision numbers have an 8-byte boundary.	Start all variables at the appropriate boundary, e.g., start floating points at a 4-byte boundary, and double precisions at an 8-byte boundary. In general, be careful when working with variables that are smaller than their boundaries.
Makefiles don't work.	UNIX System V has smaller hash tables.	Break up your Makefile into smaller sections.
Core dump while parsing feedback buffers.	<code>xfpt</code> (feedback) does not work properly due to byte alignment boundaries.	This is corrected in the 4D1-3.0 release.
During linking, you see an error message similar to this one: Bad -G num value	Your program is very large.	Add -G 0 to all modules and libraries when compiling and linking. This may slow performance slightly.
Your program aborts and reports an I/O error on a read.	FORTTRAN file structures are different.	Run the program <code>uconv(3)</code> on your binary files.
FORTTRAN program cannot find any input. You see error messages such as End of file or write to unknown unit .	Default unit numbers for FORTRAN now follow the industry standard.	Change your code so it recognizes that <code>stdin = 5</code> , <code>stdout = 6</code> , and <code>stderr = 0</code> .

While linking code that uses interlanguage calls, the linker can't find the routines that are not written in the primary language.	Interlanguage calls work differently.	See Chapter #, <i>Interlanguage Calls</i>
You see this error message: devport: command not found.	devport is no longer supported.	To configure software for a new hardware device (e.g., dial and button box), edit <i>/etc/inittab</i> . See the installation document that you received with the hardware device.
You see this error message: ranlib: command not found.	<i>ranlib</i> is no longer supported.	<i>ranlib</i> is not needed. If the symbol table for an archive is lost, use <code>ar ts lib.a</code> .
You run out of disk space.	IRIS-4D Series workstations are RISC machines. Because of this, object and executable files are as much as 1.5 times larger.	Check the amount of disk space your program uses on a 3000 Series workstation. If it just fits, you need a larger disk for your IRIS 4D.
gettp doesn't work; you cannot find out the size and position of a textport.	gettp works correctly only after you use <code>textport</code> to set the position.	You must explicitly set the size and position of textports.

Table 7-2. Language and Compiler Tips

Performance

Problem	Cause	Solution
Animated objects with shaded surfaces move slowly.	The default argument to <code>shademodel</code> is GOURAUD, not FLAT.	Change GOURAUD to FLAT in all appropriate uses of <code>shademodel</code> .
Some graphics routines run more slowly than you expect.	The new hardware doesn't improve the speed of a small group of graphics routines.	If your program uses <code>mapcolor</code> , <code>depthcue</code> , <code>gouraud</code> , <code>writemask</code> , or <code>backface</code> , you may want to use different routines to achieve the same effect.
The operating system response is very slow when running an application.	Over time, the more you call <code>malloc(3)</code> to request memory, the slower the response becomes.	Use <code>-lmalloc</code> when you link to improve the efficiency of your pointers.
Loading software options takes much longer.	The IRIS-4D Series workstations have a new installation environment that is more robust, but is also slower.	If it's appropriate, you could install the new software on only one workstation, make a tape of its filesystem, and rebuild other workstations' filesystems from this tape.

Table 7-3. Performance Tips

Hardware

Problem	Cause	Solution
Old serial cables don't fit.	Serial ports on IRIS-4D Series workstations take 9-pin connectors, while the Series 3000 workstations take 25-pin connectors.	You need either a 25 to 9 pin adaptor, or new cables.
After installing the dial and button box or digitizer tablet, the device does not respond.	It's easy to install the cable from the device to the workstation backwards.	Try reversing the cable.
	If you tried to configure the device using <code>devport</code> , it will not work. <code>devport</code> is no longer supported.	Edit <code>/etc/inittab</code> . See the document that came with the device.
	Sometimes the daemon has a false start.	Kill and restart the daemon.
Numeric keypad keys or break key do not respond.	The keyboard and the key mappings are different.	See (X-ref to 4Sight document).

Table 7-4. Hardware Tips

Section 2:

IRIS 3000 to 4D Conversion Tutorial

Contents

1. IRIS-4D Graphics Conversion	1-1
1.1 Approach to Porting to the IRIS-4D	1-2
2. Window Manager Programming	2-1
2.1 User Interface	2-1
2.2 Programming with <i>ginit</i>	2-2
2.3 Porting a Standalone Program to the Window Manager	2-6
2.3.1 Constraining Window Characteristics	2-6
2.3.2 Establishing New Window Constraints	2-8
2.3.3 Explore sample programs	2-8
2.4 Processing the Event Queue	2-19
2.4.1 Redrawing a window	2-19
2.4.2 Change in status of input devices	2-20
2.4.3 Explore a sample program	2-21
2.5 Porting Multi-Window, Double Buffered Code	2-21
2.5.1 Swapping Buffers	2-21
2.5.2 Explore sample programs	2-23
2.5.3 Processing REDRAW	2-23
2.5.4 Processing INPUTCHANGE	2-24
2.5.5 Explore a sample program	2-25
3. Resolution and Aspect Ratio of the Screen	3-1
3.1 Viewing Subroutines	3-1
3.1.1 Aspect Ratio	3-2
3.2 Window Constraints	3-3
3.2.1 Restricting Aspect Ratios	3-4
3.2.2 Specifying Window Locations	3-4
3.3 Input	3-5
3.4 Raster Data	3-5
4. Shading Polygons	4-1
4.1 Gouraud Shading on the IRIS 2000/3000	4-1
4.2 Gouraud Shading on the IRIS-4D	4-2
4.2.1 Increasing the Speed of Polygon Fill	4-2
4.2.2 Graphics Library Routines for Gouraud Shading	4-3

4.3 Gouraud Shaded Polygons in RGB Mode	4-5
5. Color and Drawing Modes	5-1
5.1 Drawing Modes	5-2
5.2 Overlays	5-2
5.2.1 Creating Overlays on the IRIS 2000/3000	5-3
5.2.2 Creating Overlays on the IRIS-4D	5-5
5.2.3 Setting the Color Map for Overlays	5-7
5.3 Underlays	5-8
5.3.1 Creating Underlays on the IRIS 2000/3000	5-8
5.3.2 Creating Underlays on the IRIS-4D	5-10
5.4 Gamma Correction	5-12
5.5 Hidden Surface Removal	5-14
5.5.1 Using the Z-buffer in Double Buffer Mode	5-15
5.5.2 Using the Z-buffer in RGB Mode	5-15
6. Cursors	6-1
6.1 Porting Simple Cursors	6-2
6.1.1 Using <code>defcursor</code>	6-2
6.2 Cross-hair Cursor	6-4
6.3 Cursors and The Window Manager	6-4
7. RGB Mode Capabilities	7-1
7.1 RGB Mode in the Window Manager	7-2
7.2 Double Buffer and RGB Modes	7-3
7.3 Depth Cueing in RGB	7-3
7.4 Supporting Multiple Display Modes	7-4
7.5 Other Features Working with RGB	7-4
8. Multiple Windows Per Process	8-1

List of Tables

Table 4-1.	Comparison of Old and New Shading Subroutines	4-4
Table 5-1.	Comparison of Gamma Correction	5-13
Table 7-1.	RGB Mode in the Window Manager	7-2

List of Figures

Figure 2-1.	A Window Manager REDRAW Event is Received	2-19
Figure 2-2.	Swapping Buffers under mex on the IRIS 2000/3000	2-22
Figure 2-3.	Swapping Buffers on One Window on the IRIS-4D	2-22
Figure 3-1.	Aspect Ratio Distortion Resulting from Different Screen Resolutions	3-2
Figure 5-1.	Flight Simulator Gauges	5-3
Figure 5-2.	house and car are drawn in separate bitplanes	5-4
Figure 5-3.	House drawn as an overlay over the car	5-4
Figure 5-4.	House in Overlay Bitplane Overlaps Car in Standard Bitplane	5-6
Figure 5-5.	house drawn as an underlay beneath the car	5-9
Figure 5-6.	On the IRIS-4D, house in underlay bitplane is drawn beneath the car in a standard bitplane	5-10

1. IRIS-4D Graphics Conversion

This tutorial is designed to assist programmers in porting graphics code from the IRIS Series 2000 and 3000 workstation to the IRIS-4D Series workstations using subroutines and techniques that are common to all members of the IRIS 4D family. It does not show you how to tune your code for a particular IRIS-4D. Once you complete the port that this book describes, see *Tuning Graphics Code for Your IRIS-4D*.

This tutorial is designed for programmers with substantial experience programming with the IRIS Graphics Library. It covers these topics:

- using and programming in the window manager (special emphasis is put on the 4D window manager features)
- understanding the effect of changes in the screen resolution
- shading polygons
- using drawing modes, overlays, underlays, gamma correction, and hidden surface removal
- controlling the cursor
- using RGB mode
- using multiple windows per process

Programming examples illustrate many of the topics, some of which compare Series 2000/3000 and 4D code.

1.1 Approach to Porting to the IRIS-4D

The main differences between programming on the IRIS 2000 and 3000 workstations and the IRIS-4D workstations are explained below. Go through this checklist and ask yourself these questions about your current IRIS 2000/3000 application:

- Does my IRIS 2000/3000 program use the window manager? If not, do I rewrite it for the window manager? If my program already runs on the IRIS 2000/3000 window manager, what changes do I make to run it on the 4D?
- How do the 4D screen resolution and aspect ratio (which are different from the 2000 and 3000) cause program changes?
- Do I use Gouraud shading in my application? Can I use `shademodel` to optimize the performance of my code on the IRIS-4D?
- How do I use the 4D routines `overlay`, `underlay`, and `drawmode` instead of `writemask` to create simple overlays, underlays, cursors, and pop-up menus.
- Do I want to use RGB mode for my application? On the 4D, RGB mode is compatible with other display modes and the window manager. Using RGB mode provides more colors than using color map mode and permits Gouraud shading and depth-cueing without loading color ramps.
- Are there user-defined cursors in my program? Do I need to use more complex cursor features such as multi-color cursors?
- How many colors are available on the IRIS-4D? The window manager on the IRIS 2000/3000 reduces the number of available colors by 75 percent. The window manager on the non-GT 4D workstations reduces the number of available colors by a mere 6.25 percent; on the GT there is no reduction.
- Does my IRIS 2000/3000 program use hidden surface removal? How can I take advantage of the increased speed of Z-buffering?
- How can I take advantage of the IRIS-4D capability to create filled concave polygons? What are the limits of concave polygon fill on the 4D?
- Does my program use feedback to use the Geometry Pipeline for matrix multiplication? In particular, does it use `xfpt()`?

This tutorial examines and contrasts some sample programs and code fragments that use features that changed from the IRIS 2000/3000 series and the IRIS-4D.

2. Window Manager Programming

4Sight, the IRIS windowing system, is always active when you are logged in to the IRIS-4D. 4Sight is the default operating environment and cannot be killed without logging out. The 4D pop-up menu interface has been changed from the IRIS Series 2000/3000.

On the Series 2000/3000, many applications have been written to run without using the window manager. On the 4D, a program that uses `ginit` or `gbegin` creates a window the same size as the screen. However, porting your application to the window manager will make it more robust and portable.

This chapter covers three topics:

- changes to the pop-up menu interface
- porting a program to the window manager
- changes to the window manager that affect programming, particularly double buffer mode graphics

2.1 User Interface

You use the mouse to control the orientation and activity of windows. The mouse brings up pop-up menus from which you can exercise a series of operations on the windows. The 4Sight interface is described at length in *Getting Started with the IRIS-4D Series Workstation*.

The configuration of the window manager user interface (the functions that are controlled from the keyboard and mouse) are specified from a file called *.mexrc*. On the Series 2000/3000, when the window manager is initiated, it searches for *.mexrc* in this order:

1. It looks for *.mexrc* in the current directory.
2. If it does not find this file, it looks for *.mexrc* in the user's home directory.
3. If it does not find this file, the window manager uses the file */usr/lib/gl2/mexrc* to define the user interface configuration.

On the IRIS-4D, 4Sight is started when the user logs in, and it uses a different configuration file, *user.ps*, to set up the user interface environment. It searches for *user.ps* in this order:

1. It looks for *user.ps* in the current directory.
2. It looks for *user.ps* in */usr/NeWS/lib*.

To create your own, customized version of *user.ps*, copy */usr/NeWS/lib/user.ps* into your home directory, and make changes to this copy. Then log out, and log in again to see the changes take effect.

A person communicates with the UNIX operating system by typing commands into a text window. On the IRIS-4D, you type these commands in *wsh*, the new window shell. *wsh* is controlled by a set of pop-up menus that are different from the Series 2000/3000. For more information on the 4Sight interface, see *Getting Started with the IRIS-4D Series Workstation*; for detailed information on customizing *user.ps*, see the *4Sight User's Guide Volume 1*, Section 3, "Programming the GL Interface".

2.2 Programming with *ginit*

On the 4D, you can still run a graphics program that uses *ginit* and is not designed to run under the window manager. *ginit* acts as if a window, which is the size of the entire screen, is opened.

To port a program to the window manager, begin with a standalone program, such as *car.c* below, that shows a moving car and a stationary house. This standalone program compiles and runs on both the IRIS Series 2000/3000 and the IRIS-4D.

■ C Program: Car.c

```
/*      car.c--IRIS 2000/3000 or IRIS-4D
 * This is the basic program which draws a car and a house.
 * This program runs on both the IRIS 2000/3000 and IRIS-4D.
 * On the 4D, its performance is not optimized. When the car
 * drives behind the house, the house is drawn on top. This
 * version does not use the window manager or overlays.
 *
 * In subsequent programs, we examine steps needed to port a
 * program to the window manager on both the IRIS 2000/3000
 * and IRIS-4D. In several cases, a corresponding IRIS
 * 2000/3000 implementation of a feature is compared with the
 * IRIS-4D.
 */

#include "gl.h"
#include "device.h"

main () {
    int    oldx,oldy,dx,dy;
    int    x,y;

    initialize ();
    dx = XMAXSCREEN / 2;
    dy = YMAXSCREEN / 2;
    while (TRUE) {
/* check polled and queued input. In this example, input is
 * only polled: getvaluator and getbutton.
 */
        if (getbutton (ESCKEY)) {
            gexit ();
            exit (0);
        }
        oldx = x;
        oldy = y;
        x = getvaluator (MOUSEX);
        y = getvaluator (MOUSEY);
        if (getbutton (LEFTMOUSE)) {
            dx = dx + (x - oldx);
            dy = dy + (y - oldy);
        }
    }
}
```

```

        drawscene (dx, dy);
        swapbuffers ();
    }
}
/*
 * initialize graphics, display mode, color map, cursors,
 * polygon fill patterns, and anything else that only needs
 * to be called once.
 */

initialize () {
    ginit ();
    doublebuffer ();
    gconfig ();
}

drawscene (x, y)
int      x,y;
{
    color (BLACK);
    clear ();
    drawcar (x, y);
    drawhouse ();
}

/* draw a car with several colors. The front window is drawn
 * and then flipped over (scaled) for the rear window. The
 * translate routine moves the car to an (x,y) position.
 */

drawcar (x, y)
int      x,y;
{
    float  fx,
           fy;

    fx = (float) x;
    fy = (float) y;
    pushmatrix ();
    translate (fx, fy, 0.0); /* move to mouse location */
    color (BLUE);           /* wheels */

```

```

    circfi (-75, -75, 20);
    circfi (75, -75, 20);
    color (RED);          /* car body */
    pmv2i (-150, -50);
    pdr2i (-125, 0);
    pdr2i (125, 0);
    pdr2i (150, -50);
    pclos ();
    color (YELLOW);       /* front window */
    drawwindow ();
    color (GREEN);        /* rear window */
    scale (-1.0, 1.0, 1.0);
    drawwindow ();
    popmatrix ();
}

/* draw a window for the car */

drawwindow () {
    pmv2i (0, 0);
    pdr2i (0, 50);
    pdr2i (50, 50);
    pdr2i (75, 0);
    pclos ();
}

/* draw a house in two colors at a fixed position, specified
 * by the translate routine.
 */

drawhouse () {
    pushmatrix ();
    translate (200.0, 100.0, 0.0); /* move house into position */
    color (MAGENTA);          /* roof */
    pmv2i (0, 0);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pdr2i (350, 0);
    pclos ();
    color (CYAN);             /* 1st floor */
    pmv2i (175, 400);

```

```

pdr2i (0, 250);
pdr2i (350, 250);
pclos ();
popmatrix ();
}

```

2.3 Porting a Standalone Program to the Window Manager

A graphics program that runs under the window manager:

- redraws a window which becomes uncovered
- directs graphics to one or several windows on the screen
- can automatically attach input devices to a window

It is recommended that IRIS-4D graphics programs run under the window manager to handle all situations with overlapping windows. Porting a standalone program to the window manager involves two key steps:

- Initializing the window or windows. Prior to opening a window, specify any constraints on the size, shape, or position of the window. Then open it.
- Processing events that signal themselves by sending an entry to the device queue. Such events can be requests to redraw the contents of a window or to make input available to a process.

2.3.1 Constraining Window Characteristics

When programming without the window manager, call `ginit` before any graphics commands are executed. In the window manager, call `winopen` to initialize a window. Call only Graphics Library routines that define or restrict the size, shape and position of a window are prior to `winopen`.

For example, to limit the minimum or maximum size (in pixels) of a graphic window, use these subroutines.

```
minsize (width, height);  
maxsize (width, height);
```

To ensure the shape (aspect ratio) of a graphics window, use this subroutine.

```
keepaspect (width, height);
```

To fix the size of a graphics window, but still be able to move it around, use this subroutine.

```
prefsize (width, height);
```

To open a graphics window and fix the location of it on the screen, use this subroutine.

```
prefposition (left, right, bottom, top);
```

These constraints are applied the first time a window is opened. The measures of the widths and heights do **not** include any banner created with `wintitle`. There are other routines which are also called before `winopen`: `stepunit`, `fudge`, `imakebackground`, `noport`, and `foreground`. Refer to the *Graphics Library User's Guide* or the *IRIS-4D GT User's Guide* for more details on these routines.

To create a window with corners at (100, 300) and (500, 600), use these subroutines:

```
prefposition (100, 500, 300, 600);  
gid = winopen ("name");
```

gid is the graphics window identifier, which is a unique integer name associated with each window. No two windows can have the same *gid*, even if the windows are created by different programs. This identifier becomes important when creating multiple windows per process.

2.3.2 Establishing New Window Constraints

Once a window is opened, you can move or resize a window by using the title bar pop-up menu or the middle mouse button. The routines called before `winopen` are saved in a list, that is used for subsequent changes to the window. For example, if `prefsize` limits the size of the window, then you can move the window around but not modify its size. If `prefposition` fixes the position of the window, then you cannot move the window at all.

To modify the existing list of constraints, use `winconstraints`. Constraint routines made after `winopen` are saved in a list. When `winconstraints` is called, the new list of constraints replaces the old list. For example, start with a window with corners at (100, 300) and (500, 600).

```
prefposition (100, 500, 300, 600);  
gid = winopen ("name");
```

To create a new list, call this additional sequence of subroutines:

```
keepaspect (4, 3);  
minsize (400, 300);  
winconstraints ();
```

`keepaspect` and `minsize` form a new list of window constraints. `winconstraints` makes the new list the official constraints list, replacing the previous `prefposition`. Now you can resize the window, but the aspect ratio remains constant and the window is never smaller than 400 pixels wide by 300 pixels high. To remove all constraints from a window, call `winconstraints()` with no window specifications preceding it.

2.3.3 Explore sample programs

This section shows how the code example, *car.c*, is ported to the window manager on both the Series 2000/3000, and the 4D. The file *wincar.c* is the window manager version of *car.c* for the IRIS 2000/3000; *win4car.c* is the version for the IRIS-4D. Later sections reexamine these programs to see

different characteristics of the window manager. In each example, pay special attention to:

- Initializing the window constraints
- Opening the window
- Creating a new list of window constraints

■ C Program: wincar.c

```
/*      wincar.c--IRIS 2000/3000
 *   The car.c program is modified to run in the window
 *   manager. Later, this code will be converted to features
 *   of the IRIS-4D.
 */

#include "gl.h"
#include "device.h"

main () {
    int    attached;
    int    oldx,
           oldy,
           dx,
           dy;
    int    dev,
           x,
           y;
    short  val;
    static int  originx,
               originy;

    initialize ();
    attached = 1;
    getsize (&x, &y);
    dx = x / 2;
    dy = y / 2;

    while (TRUE) {
        while (qtest ()) {
            dev = qread (&val);
            switch (dev) {
```

```

case REDRAW: /* window is moved or reshaped */
    reshapeviewport ();
    frontbuffer (TRUE);
    drawscene (dx, dy);
    frontbuffer (FALSE);
    break;
case INPUTCHANGE: /* user attaches or detaches
    * input focus */
    attached = (int) val;
    if (attached == 0) {
        frontbuffer (TRUE);
        drawscene (dx, dy);
        frontbuffer (FALSE);
    }
    break;
case ESCKEY:
    gexit ();
    exit (0);
    break;
default:
    break;
}
if (!attached)
while (!qtest ())
    swapbuffers ();
}
if (attached != 0) {
    oldx = x;
    oldy = y;
    x = getvaluator (MOUSEX);
    y = getvaluator (MOUSEY);
    if (getbutton (LEFTMOUSE)) {
        dx = dx + (x - oldx);
        dy = dy + (y - oldy);
    }
    drawscene (dx, dy);
    swapbuffers ();
}
}
}

```

```

/* initialize graphics for the window manager by using
 * winopen.  prefposition() is used to perfectly center a 750
 * by 560 window.  Once the window is opened, it can be
 * moved, but minsize() specifies a size limit, and
 * keepaspect() fixes an aspect ratio to the changing
 * window.  winattach() immediately attaches the input
 * devices to the window, without going through the pop-up
 * menu.
 */

```

```

initialize () {
    prefposition ((XMAXSCREEN - 750) / 2,
                  (XMAXSCREEN + 750) / 2,
                  (YMAXSCREEN - 560) / 2,
                  (YMAXSCREEN + 560) / 2);
    winopen ("car");
    minsize (750, 560);
    keepaspect (XMAXSCREEN, YMAXSCREEN);
    winconstraints ();
    doublebuffer ();
    gconfig ();
    winattach ();
}

```

```

/* The default projection transformation is based on the size
 * of the window.  The default projection could also be
 * determined by these calls:
 * getsize (&x, &y);
 * ortho2 (-0.5, x - 0.5, -0.5, y - 0.5);
 */
    qdevice (REDRAW);
    qdevice (INPUTCHANGE);
    qdevice (ESCKEY);
}

```

```

drawscene (x, y)
int      x,
        y;
{
    color (BLACK);
    clear ();
    drawcar (x, y);
}

```

```

    drawhouse ();
}
/* draw a car with several colors. The front window is drawn
 * and then flipped over (scaled) for the rear window. The
 * translate routine moves the car to an (x,y) position.
 */

drawcar (x, y)
int      x,
         y;
{
    float  fx,
           fy;

    fx = (float) x;
    fy = (float) y;
    pushmatrix ();
    translate (fx, fy, 0.0); /* move to mouse location */
    color (BLUE);           /* wheels */
    circfi (-75, -75, 20);
    circfi (75, -75, 20);
    color (RED);            /* car body */
    pmv2i (-150, -50);
    pdr2i (-125, 0);
    pdr2i (125, 0);
    pdr2i (150, -50);
    pclos ();
    color (YELLOW);         /* front window */
    drawwindow ();
    color (GREEN);          /* rear window */
    scale (-1.0, 1.0, 1.0);
    drawwindow ();
    popmatrix ();
}

/* draw a window for the car */

drawwindow () {
    pmv2i (0, 0);
    pdr2i (0, 50);
    pdr2i (50, 50);
}

```

```

    pdr2i (75, 0);
    pclos ();
}

/* draw a house in two colors at a fixed position, specified
 * by the translate routine.
 */

drawhouse () {
    pushmatrix ();
    translate (200.0, 100.0, 0.0); /* move house into position */
    color (MAGENTA);      /* roof */
    pmv2i (0, 0);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pdr2i (350, 0);
    pclos ();
    color (CYAN);      /* 1st floor */
    pmv2i (175, 400);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pclos ();
    popmatrix ();
}

```

■ C Program: win4car.c

```

/*      win4car.c--IRIS-4D
 *      The win4car.c program is modified to run in the window
 *      manager.  On the 4D, there are three differences:
 *          1) double buffering
 *          2) screen resolution
 *          3) lack of initial REDRAW token
 */

#include "gl.h"
#include "device.h"

/* On the IRIS 3000, a double buffered window manager program
 * must swap buffers, even when the input devices are not
 * attached to the window.  On the IRIS-4D, each window

```

```

* controls its own display mode independently.
*/

main () {
    int    attached;
    int    oldx,
           oldy,
           dx,
           dy;
    int    dev,
           x,
           y;
    short  val;

    initialize ();
    attached = 1;
    getsize (&x, &y);
    dx = x / 2;
    dy = y / 2;

/* process polled and queued input.
* The key difference from the 3000 is how this code waits if
* the input focus is not attached to the program. The while
* (qtest()) loop now also examines whether the input devices
* are attached. If they are not attached, the qread() will
* block, waiting for an INPUTCHANGE token to reattach and
* awaken the program.
*
* Also, you do not have to draw a still image into both
* front and back buffers. On the 4D, a graphics program
* does not have to swap between buffers when it is
* unattached.
*/

    while (TRUE) {
/* On the 3000, this code was only:
* while (qtest ()) {
*/
        while (qtest () || !attached) {
            dev = qread (&val);
            switch (dev) {

```

```

    case REDRAW: /* window is moved or reshaped */
        reshapeviewport ();
        drawscene (dx, dy);
        swapbuffers ();
        break;
    case INPUTCHANGE:
        /* user attaches or detaches
         * input focus */
        attached = (int) val;
        break;
    case ESCKEY:
        gexit ();
        exit (0);
        break;
    default:
        break;
}

/* On the 3000, while input devices were not attached to the
 * program, you swap buffers here:
 * if (!attached)
 *     while (!qtest ())
 *         swapbuffers ();
 * but on the IRIS-4D, you don't. To learn how to detach the
 * input devices, see how the INPUTCHANGE event is handled.
 */

}

if (attached != 0) {
    oldx = x;
    oldy = y;
    x = getvaluator (MOUSEX);
    y = getvaluator (MOUSEY);
    if (getbutton (LEFTMOUSE)) {
        dx = dx + (x - oldx);
        dy = dy + (y - oldy);
    }
    drawscene (dx, dy);
    swapbuffers ();
}
}
}

```

```

/* initialize graphics for the window manager by using
 * winopen. On the 3000, a 750 by 560 window is drawn, which
 * approximates the aspect ratio for the 3000 screen. On the
 * IRIS-4D, the aspect ratio is different. On the 4D, a
 * window of 750 by 600 more closely matches the aspect ratio
 * of the screen.
 *
 * On the 3000, a REDRAW token would be entered into the event
 * queue of a program as it was opened. On the 4D, no
 * REDRAW token is issued. To ensure this REDRAW token is not
 * missed, you can:
 *   1) use qenter to force a REDRAW event onto the queue
 *   2) initial all values that would have relied upon the
 * token. In this example, a REDRAW token is forced onto the
 * queue.
 */

```

```

initialize () {
    preposition ((XMAXSCREEN - 750) / 2,
                (XMAXSCREEN + 750) / 2,
                (YMAXSCREEN - 600) / 2,
                (YMAXSCREEN + 600) / 2);
    winopen ("car");
    minsize (750, 600);
    keepaspect (XMAXSCREEN, YMAXSCREEN);
    winconstraints ();
    doublebuffer ();
    gconfig ();
    winattach ();

/* The default projection transformation is based on the size
 * of the window. The default projection could also be
 * determined by these calls:
 *   getsize (&x, &y);
 *   ortho2 (-0.5, x - 0.5, -0.5, y - 0.5);
 */
    qdevice (REDRAW);
    qdevice (INPUTCHANGE);
    qdevice (ESCKEY);

    qenter (REDRAW, 0);

```

```

}

drawscene (x, y)
int      x,
        y;
{
    color (BLACK);
    clear ();
    drawcar (x, y);
    drawhouse ();
}

/* draw a car with several colors. The front window is drawn
 * and then flipped over (scaled) for the rear window. The
 * translate routine moves the car to an (x,y) position.
 * This code is not different on the IRIS 3000.
 */

drawcar (x, y)
int      x,
        y;
{
    float  fx,
          fy;

    fx = (float) x;
    fy = (float) y;
    pushmatrix ();
    translate (fx, fy, 0.0); /* move to mouse location */
    color (BLUE);           /* wheels */
    circfi (-75, -75, 20);
    circfi (75, -75, 20);
    color (RED);            /* car body */
    pmv2i (-150, -50);
    pdr2i (-125, 0);
    pdr2i (125, 0);
    pdr2i (150, -50);
    pclos ();
    color (YELLOW);         /* front window */
    drawwindow ();
    color (GREEN);         /* rear window */

```

```

    scale (-1.0, 1.0, 1.0);
    drawwindow ();
    popmatrix ();
}

/* draw a window for the car */

drawwindow () {
    pmv2i (0, 0);
    pdr2i (0, 50);
    pdr2i (50, 50);
    pdr2i (75, 0);
    pclos ();
}

/* draw a house in two colors at a fixed position, specified
 * by the translate routine.
 * This code is not different on the IRIS 3000.
 */

drawhouse () {
    pushmatrix ();
    translate (200.0, 100.0, 0.0);/* move house into position */
    color (MAGENTA);          /* roof */
    pmv2i (0, 0);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pdr2i (350, 0);
    pclos ();
    color (CYAN);             /* 1st floor */
    pmv2i (175, 400);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pclos ();
    popmatrix ();
}

```

2.4 Processing the Event Queue

The window manager communicates events by sending signals to the graphics queue of a program. A window manager program must repeatedly check and process these signals as they arrive on the queue. Without the window manager, only changes of state to input devices (for example, a pressed mouse button) signal the queue. To port a standalone program to the window manager, you must process the event queue, keeping a careful eye for special window manager events. There are two events of special interest:

- REDRAW—a window needs to be redrawn
- INPUTCHANGE—the activity of a window has changed; the input devices have just become attached to or detached from a window

2.4.1 Redrawing a window

Under the window manager, redrawing must be processed by the graphics program. Each time a window is uncovered or moved, the system must redraw it. The window manager does not store any images or other data of covered portions of windows. Instead, when a window is uncovered or moved, a REDRAW event is entered into the queue. This REDRAW is not really input, but is a signal from the window manager to a graphics program to redraw a window. The queue receives REDRAW as the device part of the event, and the value is the gid of the window to be redrawn.

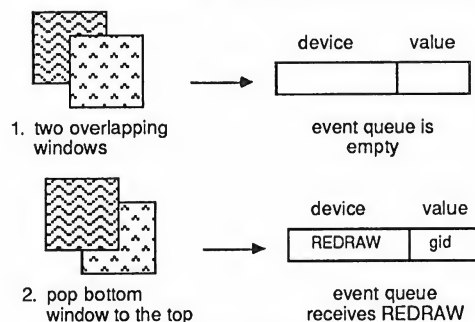


Figure 2-1. A Window Manager REDRAW Event is Received

Your program continually examines the queue to see if an input event has occurred. If something is on the queue, the first entry is removed and examined. If the device is REDRAW, then the associated value is the graphics window identifier (gid). The gid is the same number which is

returned by `winopen` when the window is originally opened. `winset` directs all graphics to the current clipping window. In case the window has been resized or moved, `reshapeviewport` reestablishes the size and position of the clipping window. Then the scene is redrawn into the window.

```
if (qtest ()) {
    dev = qread (&val);
    if (dev == REDRAW) {
        winset (val);
        reshapeviewport ();
        drawscene ();
    }
}
```

2.4.2 Change in status of input devices

The other event to check for on the queue is the `INPUTCHANGE` event. If the `INPUTCHANGE` event arrives, either you are attaching the input devices (mouse and keyboard) to operate a program or detaching (removing) those input devices. In other words, you are activating or deactivating a program.

The value that accompanies the `INPUTCHANGE` event discriminates between the two states. If the program is being deactivated, the value accompanying the `INPUTCHANGE` token is 0. If the program is being activated, the value is the graphics window identifier (`gid`).

In the example below, an ideal method for processing the `INPUTCHANGE` token is described. Declare a Logical or Boolean variable called *attached*, which always reflects the input state of the program, active or inactive. If the value accompanying the `INPUTCHANGE` is non-zero, activate the program; otherwise, deactivate it. Use the variable *attached* throughout the program to activate or deactivate sections of code.

```
if (qtest ()) {
    dev = qread (&val);
    if (dev == INPUTCHANGE) {
        if (val > 0)
            attached = TRUE;
        else if (val == 0)
            attached = FALSE;
    }
}
```

Note: In FORTRAN, Graphics Library subroutines and constants are normally shortened to the first six characters in its name. Be aware of this exception: the literal constant INPUTCHANGE is shortened to INPTCH, not INPUTC.

2.4.3 Explore a sample program

Now reexamine only the *wincar.c* program. Pay special attention to:

- the event queue
- the REDRAW event
- the INPUTCHANGE event

2.5 Porting Multi-Window, Double Buffered Code

Window manager programs in double buffer mode are processed differently than standalone programs. To port a standalone, double buffer mode program to the window manager, you must change your program to control how double buffer mode affects the window manager. Also, the window manager on the IRIS 2000/3000 manages double buffer mode differently than the IRIS-4D. To port an IRIS 2000/3000 window manager program which uses double buffer mode to the IRIS-4D, you must modify its code.

2.5.1 Swapping Buffers

On the IRIS 2000/3000, the entire screen is involved with double buffering. When one window wishes to swap buffers, it must wait for all other double buffered windows to request to swap buffers. When all windows have requested, then they are all swapped at one time. If just one double buffered program does not swap, all the other windows wait, which can hang the system.

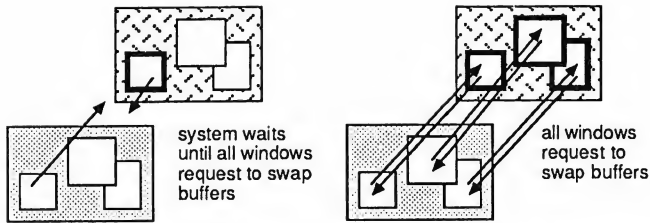


Figure 2-2. Swapping Buffers under mex on the IRIS 2000/3000

The window manager on the IRIS-4D Series workstations makes it easier to control double buffered windows. Each double buffered window is independent of any other swapping windows. Therefore, many of the programming gymnastics used to properly swap buffers on the Series 2000/3000 can be eliminated in 4D programs.

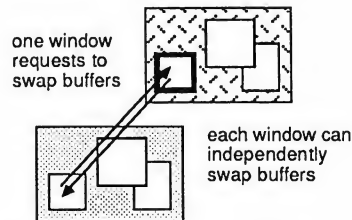


Figure 2-3. Swapping Buffers on One Window on the IRIS-4D

For example, on the Series 2000/3000, a double buffered program continually has to swap buffers, even when not attached. Otherwise, other programs are held up, waiting for the program to swap. So if the program is not attached, it checks the queue. If there are no entries on the queue (which could activate the program), it swaps buffers. The Series 2000/3000 code looks like this:

```
if (attached == FALSE) {
    while (!qtest ())
        swapbuffers ();
}
```

On the 4D, the program does not continually have to swap buffers. Each program can swap buffers independently from any other window. The constant swapping of buffers should be eliminated, and the previous code sample also should be deleted from the 4D program. Instead, when the program is not attached, it can use `qread` and wait until something enters the queue. Using `qread` is the most efficient way to deactivate a program.

The portion of code that tests and reads the queue on an IRIS Series 2000/3000 has this form:

```
while (qtest ()) {  
    dev = qread (&val);
```

To port this section of code to an IRIS-4D, change the code as follows:

```
while (qtest () || !attached) {  
    dev = qread (&val);
```

2.5.2 Explore sample programs

Now reexamine both the *wincar.c* and *win4car.c* programs. Pay special attention to how they differ:

- where buffers are swapped
- where the queue is tested

2.5.3 Processing REDRAW

To process a REDRAW event on the Series 2000/3000, the same scene is drawn into both front and back buffers. Otherwise, when the program is deactivated, and the buffers are being continually swapped, there would be a different image in each buffer and the sleeping program would appear to jump around. This example shows such 2000/3000 code:

```
dev = qread (&val);  
if (dev == REDRAW) {  
    reshapeviewport ();  
    frontbuffer (TRUE);  
    drawscene ();  
    frontbuffer (FALSE);  
}
```

One very subtle difference between the IRIS 2000/3000 and the IRIS-4D window managers is the very first event which arrives on the event queue. On the Series 2000/3000, a REDRAW event is issued for each opened window. The first time the window is opened, the program automatically receives a signal to display an image.

On the 4D, the program does not swap buffers while deactivated, so the front and back buffers do not need the same image. You only need to display the image in the visible (front) buffer. This example shows this 4D code:

```
device = qread (&val);
if (device == REDRAW) {
    reshapeviewport ();
    drawscene ();
    swapbuffers ();
}
```

Note that the final `swapbuffers` call displays the scene in the front buffer.

On the IRIS-4D, no initial event is sent to a window. Therefore, an effort must be made to draw graphics to the window, or the window remains a black rectangle. A simple way to draw graphics in the window initially is to force a REDRAW event onto the queue. This subroutine enters a REDRAW event onto the queue:

```
qenter (REDRAW, gid);
```

where `gid` is the graphics window identifier returned by the `winopen` routine.

2.5.4 Processing INPUTCHANGE

The INPUTCHANGE token notifies a program to activate or deactivate itself. On the Series 2000/3000, when a program deactivates, the same scene is drawn into both front and back buffers. Once again, this avoids a different image in each buffer which, when swapped, makes a deactivated program appear to jump around. When input devices become detached (deactivated) from a program, every window must be redrawn in both buffers:

```
dev = qread (&val);
if (dev == INPUTCHANGE) {
    if (val > 0)
        attached = TRUE
    else if (val == 0) {
        attached = FALSE
        frontbuffer (TRUE);
    }
}
```

```

        drawscene ();
        frontbuffer (FALSE);
    }
}

```

On the 4D, once a program is deactivated, swapping does not occur. It is not necessary to ensure both buffers have the same image in every window.

```

dev = qread (&val);
if (dev == INPUTCHANGE)
    if (val > 0)
        attached = TRUE;
    else if (val == 0)
        attached = FALSE;

```

2.5.5 Explore a sample program

Now again reexamine the *wincar.c* and *win4car.c* programs. Pay special attention to how they differ in processing:

- the REDRAW event
- the INPUTCHANGE event

3. Resolution and Aspect Ratio of the Screen

The screen resolution, aspect ratio, and density have changed from the IRIS 2000/3000 to the IRIS-4D. This affects many Graphics Library routines that define 2-D screen regions or 3-D viewing volumes. The density change of pixels also alters the appearance of text, polygon fill patterns, line styles, and any bitmapped, 2-D, raster data.

On the Series 2000/3000, the screen is 1024 pixels wide and 768 pixels high, an aspect ratio of 4 units of width for every 3 units of height. On the 4D, the screen is 1280 pixels wide and 1024 pixels high, an aspect ratio of 5 units of width for every 4 units of height.

3.1 Viewing Subroutines

The new resolution and aspect ratio distorts how objects are viewed. The distortion can change how objects are perceived in 3-D space or how they are projected onto the 2-D screen. You may need to change these types of Graphics Library subroutines:

- define 2-D and 3-D viewing volumes
 - `ortho (left, right, bottom, top, near, far)`
 - `ortho2 (left, right, bottom, top)`
 - `perspective (fovyangle, aspect, near, far)`
 - `window (left, right, bottom, top, near, far)`

- define 2-D screen regions for gross and fine clipping
 - viewport (left, right, bottom, top)
 - getviewport (left, right, bottom, top)
 - scrmask (left, right, bottom, top)
 - getscrmask (left, right, bottom, top)
- inversely map from screen space to viewing space
 - mapw (vobj, ix, iy, x1, y1, z1, x2, y2, z2)
 - mapw2 (vobj, ix, iy, x1, y1)

3.1.1 Aspect Ratio

The aspect ratio of the IRIS-4D monitor can distort how the 3-D viewing routines, *ortho*, *ortho2*, *perspective*, and *window* project objects onto the screen. To prevent distortion, the aspect ratio of the viewing volume should be the same as the monitor. Using the literal constants, `XMAXSCREEN` and `YMAXSCREEN`, you can avoid immutably tying your application to the Series 2000/3000 screen size. In the *gl.h* and *fgl.h* include files, `XMAXSCREEN` and `YMAXSCREEN` are defined as the number of pixels of width and height, respectively. (In FORTRAN, both constants are shortened to six character names, `XMAXSC` and `YMAXSC`.) On the IRIS 2000/3000, `XMAXSCREEN` is 1023 and `YMAXSCREEN` is 767. On the IRIS-4D, `XMAXSCREEN` is 1279 and `YMAXSCREEN` is 1023.

To avoid distortion with *perspective*, set the aspect ratio of the 3-D viewing volume to match that of the display as shown below.

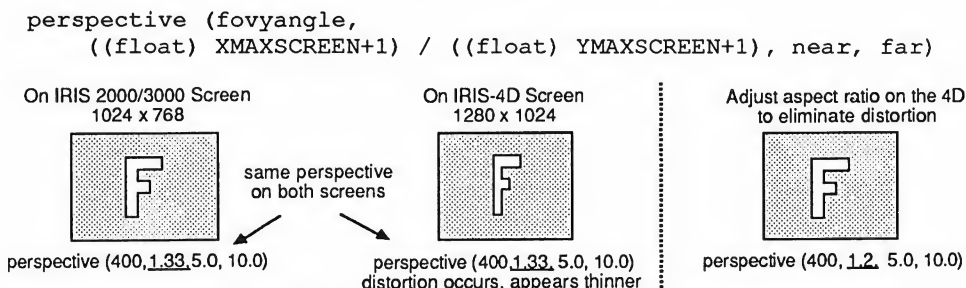


Figure 3-1. Aspect Ratio Distortion Resulting from Different Screen Resolutions

With `ortho`, `ortho2`, and `window`, which do not define the aspect ratio with one value, changing the aspect ratio is not as simple. To control the aspect ratio with these three subroutines, alter the `left`, `right`, `bottom` and `top` values.

The screen resolution change greatly affects the subroutines `viewport` and `scrmask`, which define regions on the screen. `viewport` defines the screen rectangle for clipping geometry and for the gross clipping of text. `scrmask` defines the rectangle for the fine clipping of text. Code which reads:

```
viewport (0, 1023, 0, 767)
```

should be changed to:

```
viewport (0, XMAXSCREEN, 0, YMAXSCREEN)
```

This is more flexible and is preferred over using numerical constants. Similarly, `mapw`, the inverse viewing mapping from 2-D to 3-D, and `mapw2` are affected by the change in screen resolution.

Changing the aspect ratio of viewing volumes is necessary to prevent distortion, but, unfortunately, the values are not always easy to choose.

3.2 Window Constraints

Graphics programs that run under the window manager can constrain the size, aspect ratio or position of their windows. When porting a program from the IRIS 2000/3000 to the 4D, a change in the window constraints often improves the appearance of windows on the screen. Windows that appear most harmonious:

- have the same aspect ratio as the monitor itself
- have a similar aspect ratio to other windows
- are centered on the screen
- cover the entire screen without overlapping

To achieve your desired visual effect, you may need to change these subroutines:

- `keepaspect (width, height)`
- `prefposition (left, right, bottom, top)`
- `prefsize (width, height)`
- `maxsize (width, height)`
- `winposition (left, right, bottom, top)`
- `winmove (orgx, orgy)`

3.2.1 Restricting Aspect Ratios

`keepaspect` restricts the aspect ratio of opened windows. A flexible aspect ratio, which works on both the IRIS 2000/3000 and the 4D, is specified like this:

```
keepaspect (XMAXSCREEN+1, YMAXSCREEN+1)
```

3.2.2 Specifying Window Locations

`prefposition`, `winposition`, and `winmove` specify window locations. Because the 4D screen has a larger resolution, calls to these routines may require changes. For example, the call below covers the entire screen on the IRIS 2000/3000 workstation.

```
prefposition (0, 1023, 0, 767)
```

However, it does not cover the increased resolution of the 4D screen. Wise use of the literal constants, `XMAXSCREEN` and `YMAXSCREEN`, solves this problem.

3.3 Input

Input values based on cursor position are affected by the new larger screen. On the 4D, MOUSEX and MOUSEY become as large as 1279 and 1023, respectively.

The larger screen affects input polled from the user with the `getvaluator` function and the `setvaluator` routine, or input saved in the queue for the MOUSEX and MOUSEY devices. When queueing, these values frequently are tied to a mouse button. To make a Series 2000/3000 program that uses these subroutines run on a 4D, you may need to make the following changes:

- polled input

- `value = getvaluator (device);`
 - `setvaluator (device, init, min, max);`

- queued valuator input

- `qdevice (MOUSEX);`
 - `qdevice (MOUSEY);`
 - `tie (BUTTONMOUSE, MOUSEX, MOUSEY);`
 - `device = qread (&value);`

Note: If you change the boundaries of the mouse movement with `setvaluator`, when the program is exited, the values are not restored. Your program should reset them before exiting.

3.4 Raster Data

Because of the higher screen resolution on the 4D, graphic images, pixel mapped characters and cursors, polygon fill patterns and line styles will appear smaller and finer, about three-quarters the height and width of the 2000/3000. The screen location will also affect positioning graphic images (screen dumps) and pixel mapped characters (text).

This affects these categories of Graphics Library subroutines:

- raster images
 - readpixels (number, pixelarray);
 - writepixels (number, pixelarray);
 - readRGB (number, redarray, greenarray, bluearray);
 - writeRGB (number, redarray, greenarray, bluearray);
- text
 - cmov (x, y, z);
 - charstr ("string");
 - width = strwidth ("string");
- icons and other bitmaps
 - defrasterfont (index, numchar, height, attributes, numraster, bitmap);
 - deflinestyle (index, pattern);
 - defpattern (index, size, pattern);
 - defcursor (index, icon);

4. Shading Polygons

Flat shading, which uses a single color for every filled polygon, is the fastest and simplest method to shade a surface. However, the appearance of flat shaded surfaces is not realistic. In particular, a seam forms where adjacent flat shaded polygons meet.

Gouraud shading provides a more colorful and realistic representation of a polygonal surface than flat shading. The Gouraud algorithm uses the colors at each vertex to interpolate the colors for the pixels that lie within the edges of the polygon.

4.1 Gouraud Shading on the IRIS 2000/3000

On the IRIS 2000/3000, `splf` and `spclos` are two routines that draw Gouraud shaded filled polygons. With `spclos`, `setshade` specifies the colors at the vertices of the polygons. At any time, you can use either flat shading or Gouraud shading to fill a polygon. The IRIS 2000/3000 hardware finds the fastest way possible to meet your shading request.

This code fragment shows how to draw a shaded triangle on the IRIS Series 2000/3000 workstations.

```
/*set up arrays */
int      parray[3][2] = {
    100, 100, 300, 500, 500, 100
};

unsigned short  iarray[3] = {
    8, 16, 23
};

/* make color ramp from index 8 to 23 for Gouraud shading */
for (i = 0; i < 16; i++){
```

```

    mapcolor (i + 8, 0, i * 255 / 16, i * 255 / 16);
    mapcolor (BACKGROUND, 80, 200, 200);
}
/* draw the shaded polygon */
drawpoly() {
    splf2i(3, parray, iarray);
}

```

This code fragment shows how to maintain gouraud shading on a triangle while two vertices are fixed, and the user may move the third.

```

/* make color ramp from index 8 to 23 for Gouraud shading */
for (i = 0; i < 16; i++) {
    mapcolor (i + 8, 0, i * 255 / 16, i * 255 / 16);
    mapcolor (BACKGROUND, 80, 200, 200);
}
drawpoly (x, y)
int      x,
        y;
{
    setshade (8);
    pmv2i (100, 100);
    setshade (16);
    pdr2i (300, 500);
    setshade (23);
    pdr2i (x, y);
    spclos ();
}

```

4.2 Gouraud Shading on the IRIS-4D

On IRIS-4D Series workstations, Gouraud shading is the default shading technique. You can use the new `shademodel` subroutine to change the default in order to speed up the rate of polygon filling.

4.2.1 Increasing the Speed of Polygon Fill

On the IRIS-4D, the shading hardware (in the rendering subsystem) operates in two different modes: `FLAT` and `GOURAUD`. In `FLAT`, flat shading is optimized, but the system may become incapable of Gouraud shading. In `GOURAUD`, polygon fill is optimized for Gouraud shading and flat shading is slower.

`shademodel` switches between shading modes on the IRIS-4D. The default mode is `GOURAUD`, in which the shading hardware is optimized for Gouraud shading. To achieve the highest performance on non-GT workstations, use Gouraud shading only when it is necessary. To switch the polygon fill mode to `FLAT`, add this line of code:

```
shademodel (FLAT) ;
```

In the majority of cases, follow these guidelines:

- Initially switch the polygon fill mode to `FLAT` until Gouraud shading is actually requested.
- Switch to `GOURAUD` just prior to drawing Gouraud shaded polygons.
- Return to `FLAT` when drawing Gouraud shaded polygons is finished.
- Avoid unnecessary `shademodel` calls, as they may slow performance.

4.2.2 Graphics Library Routines for Gouraud Shading

The IRIS-4D Graphics Library attempts to reduce redundant subroutines. The 4D routine, `color`, denotes the color at a vertex, replacing the 2000/3000 routine, `setshade`. On the 4D, `pclos` closes a Gouraud shaded polygon, replacing `spclos`. Both `setshade` and `spclos` are included in the IRIS-4D Graphics Library for compatibility. Use `color` and `pclos`, respectively, for Gouraud shading on the IRIS-4D workstations.

Technique	2000/3000 Subroutines	IRIS-4D Subroutines
specify arrays of coordinates and colors	splf	splf
specify vertex coordinates	pmv, pdr	pmv, pdr
specify vertex colors	setshade	color
close and fill polygon	spclos	pclos

Table 4-1. Comparison of Old and New Shading Subroutines

On the GT, use the high performance subroutines `bgnpolygon...endpolygon` to specify a closed, shaded polygon. See *Tuning Graphics Code for Your IRIS-4D*.

The following code fragments illustrate different ways to use the shading subroutines.

This code uses `shademodel` and `splf` to draw a shaded triangle.

```
int      parray[3][2] = {
    100, 100, 300, 500, 500, 100
};

unsigned short  iarray[3] = {
    8, 16, 23
};

/*  make color ramp from index 8 to 23 for Gouraud shading  */
for (i = 0; i < 16; i++) {
    mapcolor (i + 8, 0, i * 255 / 16, i * 255 / 16);
    mapcolor (BACKGROUND, 80, 200, 200);
}

/*  The shademodel is set to FLAT for the screen clear, but it
 *   is set to GOURAUD to draw the shaded polygon.  After the
 *   polygon is drawn, it is reset to FLAT.  The (x, y)
 *   position is the location for the third vertex.
 */

shademodel (FLAT);
color (BACKGROUND);
clear ();
shademodel (GOURAUD);
```

```

parray[2][0] = x;
parray[2][1] = y;
drawpoly ();
shademodel (FLAT);

/* Use splf() to draw the shaded polygon */
drawpoly () {
    splf2i (3, parray, iarray);
}

```

This code uses `color`, `pmv`, `pdr` and `pclos` since `shademodel` takes care of the shading. Compare this to the code fragment in Section 4.1.

```

drawpoly (x, y)
int      x,
        y;
{
    color (8);
    pmv2i (100, 100);
    color (16);
    pdr2i (300, 500);
    color (23);
    pdr2i (x, y);
    pclos ();
}

```

4.3 Gouraud Shaded Polygons in RGB Mode

The IRIS 2000/3000 only interpolate color map indices for Gouraud shading. The 4D shading hardware can interpolate the red, green and blue components in RGB mode. Gouraud shading in RGB mode works only with `pclos`. At every vertex of the polygon, an `RGBcolor` is stated.

If you use Gouraud shading in RGB mode, you:

- do not have to load the color map with a ramp of colors.
- get more colors and smoother shading from RGB mode than from small section of a color map.

The distinct disadvantage to using RGB mode is that the color values used for shading are limited on some IRIS-4D Series workstations that have a limited number of bitplanes.

This code fragment shows how to draw the same shaded triangle from Sections 4.1 and 4.2 in RGB mode.

```
drawpoly (x, y)
int      x,
        y;
{
    RGBcolor (0, 0, 0);
    pmv2i (100, 100);
    RGBcolor (0, 127, 127);
    pdr2i (300, 500);
    RGBcolor (0, 255, 255);
    pdr2i (x, y);
    pclos ();
}
```

5. Color and Drawing Modes

The IRIS 2000/3000 has one set of multipurpose bitplanes, which store a variety of data: window manager pop-up menus, color values for standard images, as well as depth values for hidden surface removal.

In contrast, the IRIS-4D has several different sets of single-purpose bitplanes. There are up to 48 standard color bitplanes (on the GT), storing 48 bits of color information per pixel. There are also four overlay bitplanes. The window manager takes two, and you can program the other two for overlays or underlays. For Z-buffer hidden surface removal, 24 bits of depth bitplanes is available as an option on all workstations except for the GT, where they are standard.

Multi-mode graphics processor (MGP) custom chips accommodate different sets of bitplanes. Programs can run with any combination of single or double buffer mode, color map or RGB mode, and overlays or underlays. The different sets of bitplanes work together and do not restrict the functionality of others.

This chapter covers these topics:

- drawing modes
- overlays
- underlays
- gamma correction
- hidden surface removal

5.1 Drawing Modes

You can draw shapes into each set of bitplanes. The subroutines that draw Graphics Library primitives (rectangle, circles, lines, points and polygons) can draw shapes into all sets of bitplanes. Each set of bitplanes has its own color map, writemask, and current drawing color.

A new Graphics Library subroutine, `drawmode(mode)`, specifies the current *drawing mode*, which is the current set of bitplanes affected by Graphics Library routines. The initial drawing mode is `NORMALDRAW`, which directs graphics subroutines to the standard bitplanes. `drawmode` switches graphics subroutines between normal drawing, overlays, underlays, and cursors. Calls to `color`, `getcolor`, `writemask`, `getwritemask`, `mapcolor`, and `getmcolor` are affected by the current drawing mode.

Drawing modes include:

- `NORMALDRAW`, which sets operations for 12-bit color map mode or RGB mode
- `OVERDRAW`, which sets operations for the overlay
- `UNDERDRAW`, which sets operations for the underlay
- `CURSORDRAW`, which sets operations for the cursor
- `PUPDRAW`, which sets operations for the pop-up menu

Note: In FORTRAN the above drawing modes are `NORMDR`, `OVRDRW`, `UNDRDR`, `CURSDR`, and `PUPDRW`.

5.2 Overlays

Static overlays are often painted on top of other objects. Since these overlays are not moved, performance may be increased dramatically if the overlays are only drawn once and not redrawn for every frame. For example, the numerical scale on the gauges of the flight simulator is an overlay. The system only updates the rectangle that indicates speed or fuel consumption.

On the IRIS 2000/3000, overlays are drawn to the standard bitplanes. Overlays can use only the color map (not RGB mode), and, consequently, reduce the number of available colors.

On the IRIS-4D, a three-color overlay can be drawn to the window manager bitplanes. You can use these window manager bitplanes with RGB mode without reducing the number of available colors.

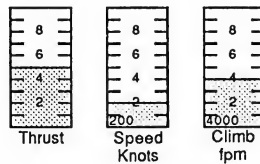


Figure 5-1. Flight Simulator Gauges

5.2.1 Creating Overlays on the IRIS 2000/3000

On the IRIS 2000/3000, you make overlays by manipulating the color map and using `writemask`. `writemask` controls whether you can write upon (read/write) bitplanes or if you only can read and view (read only) bitplane images. You can still use this technique, but the IRIS-4D Series offers an easier method.

To create an overlay image, follow these steps:

1. Draw an image to a read/write bitplane.
2. Load colors into the color map.
3. Call `writemask` before entering the main loop of a program (where you continually update and change an object) to preserve the image in the overlay bitplane.

For instance, imagine a scene with a moving car that goes behind a house. The house is a one-color static overlay. It doesn't move and covers the car. You draw the house into a bitplane. In color map mode, the bitplanes contain the numerical index into the color map for each pixel. To draw the house into bitplane 2, use color 2 for the house. Once you draw the house, the new `writemask` preserves the contents of bitplane 2. The system draws the car, clears and redraws it into bitplane 1 (with color 1) without damaging the house in the other bitplane.

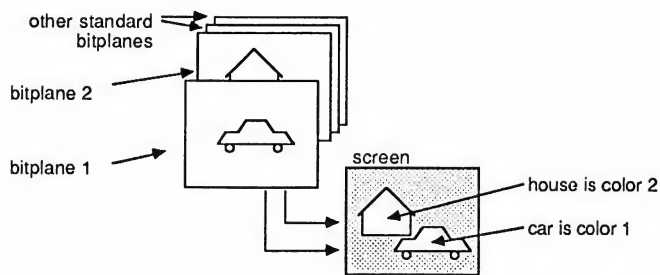


Figure 5-2. house and car are drawn in separate bitplanes

This process is only the first step. When the car and house overlap, the color index of a pixel is neither 1 (car) nor 2 (house). The color index of the pixel is $1 + 2$, i. e., 3, which is a completely different color in the map. Since the house blocks the car, the overlap pixel should appear with the same red, green and blue values as the house alone. The trick is to change the entry of the color map for the overlap (car and house) color to match the color for the house. The overlapping area blends into the house.

```
mapcolor (2, red, green, blue); /* house */
mapcolor (3, red, green, blue); /* overlap */
```

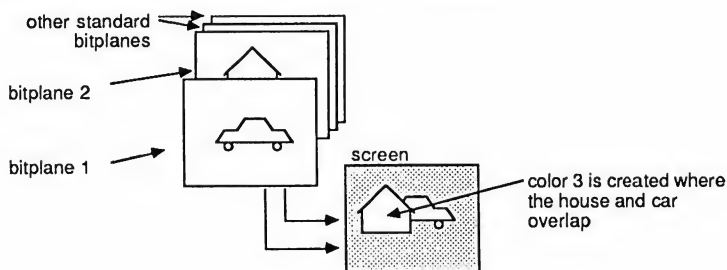


Figure 5-3. House drawn as an overlay over the car

This code fragment demonstrates the implementation of an overlay, using the standard bitplanes on the IRIS 2000/3000. The car uses four colors and is drawn into bitplanes 1, 2, and 3. The house is drawn with colors 8 and 16 into the bitplanes 4 and 5; the house is the overlay. Color map entries 8 to 15 cover every possible combination of the roof of the house overlapping any part of the car. These color map entries are loaded with the color for the roof of the house, so that when the car and roof of the house overlap, it appears as the roof. For color map entries 16 to 23, the color for the first floor of the house is loaded.

```

/* be sure that the REDRAW token redraws the overlay */
qdevice (REDRAW);

/* load the color map */
for (i = 8; i < 16; i++)
mapcolor (i, 255, 0, 255);
for (i = 16; i < 24; i++)
mapcolor (i, 0, 255, 255);

/* draw the overlay */
writemask (0xFFFF);
frontbuffer (TRUE);
color (BLACK);
clear ();
drawhouse ();
frontbuffer (FALSE);
writemask (0x7);

drawhouse () {
    pushmatrix ();
    translate (200.0, 100.0, 0.0); /* move house into position */
    color (8);                      /* roof */
    pmv2i (0, 0);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pdr2i (350, 0);
    pclos ();
    color (16);                      /* 1st floor */
    pmv2i (175, 400);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pclos ();
    popmatrix ();
}

```

5.2.2 Creating Overlays on the IRIS-4D

On the 2000/3000, static overlays (and underlays) require use of writemask. writemask eliminates the number of usable image bitplanes. The loss of each bitplane halves the number of colors. In RGB mode on the 2000/3000, you cannot make overlays or underlays.

On the 4D, two window manager bitplanes are reserved for static overlays (or underlays). These bitplanes are not shared with the standard image bitplanes, so colors are not lost.

First, declare how you plan to use the special overlay/underlay bitplanes. The two bitplanes are either used as overlays or underlays but not both at once. To initialize the bitplanes for overlays, call this code sequence once:

```
overlay (2);  
gconfig ();
```

The parameter, 2, specifies how many bitplanes are used.

Once the bitplanes are initialized, you must set the current drawing mode to OVERDRAW.

```
drawmode (OVERDRAW);
```

Then you can define colors, set a writemask, and draw an object into the two overlay bitplanes.

The multi-mode graphics processor (MGP) chips multiplex between data found in the standard bitplanes and the overlay/underlay window manager bitplanes. When in overlay drawing mode, if a zero value is found in the overlay bitplanes for each pixel, then the color specified by the standard bitplanes is drawn. If a non-zero value is in the overlay bitplanes, the MGPs choose to draw the overlay color for that pixel. The overlay bitplanes override the standard bitplanes in this simple hierarchy:

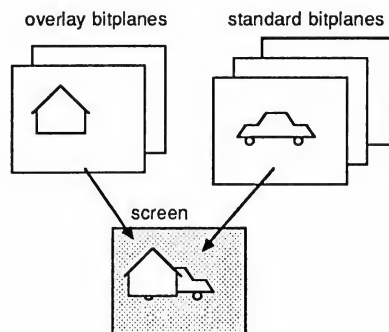


Figure 5-4. House in Overlay Bitplane Overlaps Car in Standard Bitplane

To create overlays with more than three colors, you may need more than two bitplanes. In those situations, you can use `writemask` to reserve standard bitplanes for overlays, just as is done on the IRIS 2000/3000.

5.2.3 Setting the Color Map for Overlays

There is a small color map for the overlay colors and color map entries for all non-zero overlay values. With two overlay bitplanes, three non-zero overlay colors can be formed: binary 01, 10, and 11 (or in base 10: 1, 2 and 3). Once in the overlay drawing mode, you can load the color map and specify a writemask for the two bitplanes. Any effort to load color index 0 is ignored because color 0 cannot be defined for overlays.

```
drawmode (OVERDRAW);
mapcolor (1, 0, 255, 0);
mapcolor (2, 255, 255, 0);
mapcolor (3, 255, 0, 255);
```

To draw a shape into the overlay bitplanes, choose a color index and start drawing. This sequence of code creates a green rectangle as an overlay. The final `drawmode` call restores the drawing mode to use the standard image bitplanes.

```
drawmode (OVERDRAW);
mapcolor (1, 0, 255, 0);
color (1);
rectfi (200, 200, 300, 300);
drawmode (NORMALDRAW);
```

This code fragment shows the car and house example from the IRIS 2000/3000 written for the IRIS-4D to use the overlay capability.

```
/* On the IRIS 3000, the color map would be loaded for
 * the overlays:
 *           for (i = 8; i < 16; i++)
 *               mapcolor (i, 255, 0, 255);
 *           for (i = 16; i < 24; i++)
 *               mapcolor (i, 0, 255, 255);
 */

overlay (2);
gconfig ();
/* there is no frontbuffer or backbuffer for overlay */
drawmode (OVERDRAW);
mapcolor (1, 255, 0, 255);
mapcolor (2, 0, 255, 255);
drawmode (NORMALDRAW);
drawhouse ();
}

drawhouse () {
```

```

drawmode (OVERDRAW);
color (0);
clear ();
color (1);                      /* roof */
pmv2i (0, 0);
pdr2i (0, 250);
pdr2i (350, 250);
pdr2i (350, 0);
pclos ();
color (2);                      /* 1st floor */
pmv2i (175, 400);
pdr2i (0, 250);
pdr2i (350, 250);
pclos ();
drawmode (NORMALDRAW);
}

```

5.3 Underlays

A static underlay is like an overlay, except it is painted to be a background for other objects. Using the familiar car and house example, if the house is a static underlay, the car moves in front of it.

5.3.1 Creating Underlays on the IRIS 2000/3000

On the IRIS 2000/3000, you use `writemask()` to create underlays. When the car and house overlap, the car should go in front of the house. To create an underlay image, you must complete the same steps (on page xx) as the overlay. Only the color map changes. As with underlays, you can still use this technique on the IRIS-4D Series.

To achieve the proper effect, the underlay color (3) matches the same red, green and blue values of the car.

```

mapcolor (1, red, green, blue); /* car */
mapcolor (3, red, green, blue); /* overlap */

```

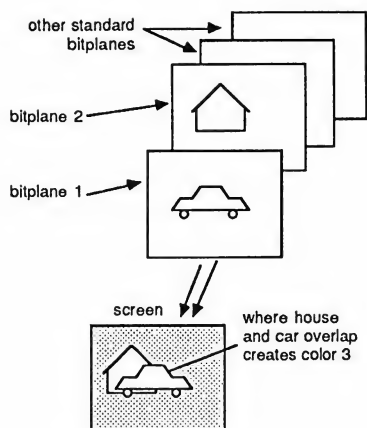


Figure 5-5. house drawn as an underlay beneath the car

Compare this IRIS 2000/3000 underlay code fragment to its counterpart overlay code fragment. Only the color map has been changed. Now when the house and car overlap, the color of the car should dominate. For example, color map entries 1, 9 (e.g., 8 + 1), and 17 (e.g., 16 + 1) are loaded with the color for the car body. When the car body overlaps the roof of the house (color 8) or the first floor of the house (color 16), it appears as the color of the car body. The house appears to lie underneath the car.

```
mapcolor (8, 255, 0, 255);          /* MAGENTA for house */
mapcolor (16, 0, 255, 255);         /* CYAN for house */
for (i = 0; i < 24; i = i + 8) {
    mapcolor (i + 1, 255, 0, 0);     /* RED for 1, 9, 17 */
    mapcolor (i + 2, 0, 255, 0);     /* GREEN for 2, 10, 18 */
    mapcolor (i + 3, 255, 255, 0);   /* YELLOW for 3, 11, 19 */
    mapcolor (i + 4, 0, 0, 255);     /* BLUE for 4, 12, 20 */
}

writemask (0xFFFF);
frontbuffer (TRUE);
color (BLACK);
clear ();
drawhouse ();
frontbuffer (FALSE);
writemask (0x7);

drawhouse () {
    color (8);                       /* roof */
    pmv2i (0, 0);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pdr2i (350, 0);
}
```

```

pclos ();
color (16);
pmv2i (175, 400);
pdr2i (0, 250);
pdr2i (350, 250);
pclos ();
}

```

5.3.2 Creating Underlays on the IRIS-4D

Creating IRIS-4D underlays is very similar to creating IRIS-4D overlays. Initialize the window manager bitplanes with this sequence:

```

underlay (2);
gconfig ();

```

The underlay mode resembles the overlay mode, but the hierarchy is different. In underlay drawing mode, the multi-mode graphics processors check the value in the standard bitplanes for each pixel. If the standard bitplanes contain a non-zero value, the processors draw the standard color for that pixel. If the standard bitplanes contain a zero value, the processors draw the color specified by the underlay bitplane. In other words, the standard bitplanes override the underlay bitplane.

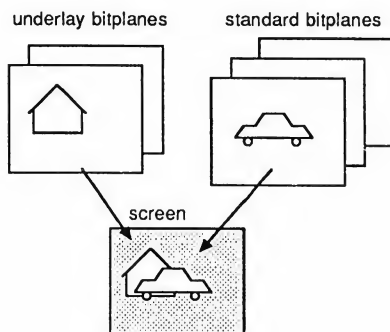


Figure 5-6. On the IRIS-4D, house in underlay bitplane is drawn beneath the car in a standard bitplane

For underlays, a zero value is accepted. Therefore, there can be as many as four underlay colors, as opposed to three for overlays. The sequence below creates a legitimate, blue rectangle underlay, and then restores the normal color mode.

```

drawmode (UNDERDRAW);
mapcolor (0, 0, 0, 255);
color (0);
rectfi (200, 200, 300, 300);
drawmode (NORMALDRAW);

```

On the IRIS-4D, two bitplanes are reserved for either overlays or underlays. You cannot simultaneously use the bitplanes for both overlays and underlays. You may need more than two bitplanes to create overlays with more than three colors, underlays with more than four colors, or simultaneous overlays and underlays. In those situations, you can still use `writemask` to reserve standard bitplanes for overlays and underlays, just as on the IRIS 2000/3000.

This is a 4D version of the car and house code fragment, using the underlay bitplanes.

```

underlay (2);
gconfig ();
drawmode (UNDERDRAW);
mapcolor (0, 0, 0, 0);
mapcolor (1, 255, 0, 255);
mapcolor (2, 0, 255, 255);
drawmode (NORMALDRAW);
drawhouse ();

drawhouse () {
    drawmode (UNDERDRAW);
    color (0);
    clear ();
    color (1);                      /* roof */
    pmv2i (0, 0);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pdr2i (350, 0);
    pclos ();
    color (2);                      /* 1st floor */
    pmv2i (175, 400);
    pdr2i (0, 250);
    pdr2i (350, 250);
    pclos ();
    drawmode (NORMALDRAW);
}

```

5.4 Gamma Correction

Color ramps are created for smooth shading or intensity depth cueing. To make a color ramp, start with a very dark shade of a color and increase its brightness. It is easiest to increase the color brightness with linear steps, but linear steps do not usually appear sharpest.

The perception of light depends upon two strong, non-linear factors:

- your eye
- the display screen (CRT)

To compensate for these non-linearities, create non-linear (gamma corrected) color ramps. As stated in Foley and van Dam, the eye is “sensitive to ratios of intensity levels rather than to their absolute values.” In general, brightness levels should be increased logarithmically.

On the 2000/3000, shading and depth cueing are effective only in color map mode. To make a gamma corrected color ramp, the color intensities loaded into the color map are gamma corrected.

On all IRIS-4D Series workstations, shading and depth cueing can be supported in RGB mode. The IRIS 4D non-GT workstations and the IRIS 4D/GT use different methods to do this. The GT does gamma correction using special, dedicated hardware, while the non-GT workstations use this method:

- In RGB mode, the actual values in the standard (RGB) bitplanes are not directly drawn to the display. The RGB values reference the highest 256 colors in the color map (indices 3840-4095).
- The color drawn to the display is the color loaded into the corresponding map index. For example, if the red value for a pixel is 200, then the color drawn to the screen is the red value stored in the color map at index 3840+200, or 4040. If a previous command, such as `mapcolor (4040, 233, GREEN, BLUE)`, has been executed, then a red value of 200 generates a displayed red value of 233.
- By default, these 256 colors are stored with a linear grey scale ramp. That is, index 3840 contains R=G=B=0, index 3841 contains R=G=B=1, and so forth, and there is no gamma correction.

- This special intensity modulation affects RGB mode graphics and the window manager color modes (pop-up menu, overlay, underlay, and cursor). Colors loaded into the standard color map are **not** affected.

Note: Gamma correction affects the top 256 indices in the color map. You should not use these colors for your application programs. If you load these colors, there is no warning, but the cursor and pop-up menus may be immediately affected. It is advised that you restrict access to these top colors with `gammaramp`.

This table summarizes the differences.

Machine/Mode	4D non-GT	4D/GT
RGB Mode	Uses top 256 colors; may affect overlays and underlays.	Uses special hardware; doesn't use top 256 colors
Color Map Mode	You create your own ramp using <code>gamma</code> and <code>mapcolor</code> ; performance may be slow.	Uses special hardware; performances increases.

Table 5-1. Comparison of Gamma Correction

The code fragment below demonstrates creating a gamma corrected color ramp. You should enter three arguments to `gamma`. The first argument is the gamma correction constant. The second and third arguments are the color map indices for the beginning and end of the ramp. Use `gammaramp` to load the 256 gamma correcting intensities on the IRIS-4D. One argument, the gamma constant, is passed to the program.

```
/* gamma_spectrum -- This routine creates a gamma-corrected
 * color ramp. gamma is the strength of gamma correction.
 * The next six arguments are the starting and finishing RGB
 * values of the ramp. All the interim color values on the
 * ramp are determined between these extremes.
 */

gamma_spectrum (gamma, brightred, brightgreen, brightblue,
               darkred, darkgreen, darkblue)
float gamma;
unsigned char brightred,
```

```

        brightgreen,
        brightblue;
unsigned char    darkred,
                darkgreen,
                darkblue;
{
    int    i;
    float  ginc;
    short  r[RAMP_SIZE],
            g[RAMP_SIZE],
            b[RAMP_SIZE];    /* gamma-corrected RGB */

    for (i = 0; i < RAMP_SIZE; i++) {
        ginc = pow ((float) i / (float) (RAMP_SIZE - 1),
                    1.0 / gamma);
        r[i] = (short) (ginc *
                        (brightred - darkred) + darkred);
        g[i] = (short) (ginc *
                        (brightgreen - darkgreen) + darkgreen);
        b[i] = (short) (ginc *
                        (brightblue - darkblue) + darkblue);
    }
    gammaramp (r, g, b);
}

```

5.5 Hidden Surface Removal

On the IRIS 2000/3000, hidden surface removal requires a tradeoff between programming effort and performance of the program. To render solid objects with hidden surfaces properly removed, you:

- sort the order that polygons are drawn from different viewpoints. This approach does not work for irregularly shaped objects (e. g., with woven polygons). Also this approach does not systematically create an order to draw the polygons so that hidden surfaces are removed.
- create a binary space partition tree to view an object with real-time motion. To implement this approach, you must write a large amount of code to create and traverse binary trees and process polygonal data.
- use the hardware Z-buffer for general and effective hidden surface removal. On the IRIS 2000/3000, the Z-buffer cannot remove hidden surfaces in real-time. In addition, the depth values for Z-buffer are kept in the standard bitplanes, which deprives using the Z-buffer in tandem with double buffer mode or RGB mode.

On the IRIS-4D, the hidden surface removal capabilities of the hardware Z-buffer are greatly improved over the IRIS 2000/3000:

- The resolution of the depth (z) values has increased from 16 to 24 bits.
- The speed of hidden surface removal is real-time.

The depth values are stored in depth bitplanes which are independent of standard and window manager bitplanes. Accordingly, use of the Z-buffer is compatible with double buffer mode and RGB mode.

The improvements in the Z-buffer can change the entire way you view geometric objects. On the 2000/3000, with its slower Z-buffer, you would:

- control a wireframe of an object in real-time
- before solidly rendering the object, switch to single buffer mode
- receive an input event (typically, press the 'z' key) to trigger use of the Z-buffer to solidly render the object in a fixed position

On the IRIS-4D, you can:

- control a solidly rendered object in real-time with hidden surfaces removed
- stay in double buffer mode

5.5.1 Using the Z-buffer in Double Buffer Mode

The Z-buffer is now compatible with double buffer mode. Therefore, animated scenes with hidden surface removal can be drawn without flickering.

If your program now uses Z-buffering, you can turn on double-buffering using `doublebuffer` and `swapbuffers`, and the program will work correctly.

5.5.2 Using the Z-buffer in RGB Mode

The Z-buffer is also compatible with RGB mode. See Chapter 7 for more information on RGB mode.

6. Cursors

Control of the cursor (the icon which follows the movement of the input device) has changed between the IRIS 2000/3000 to the IRIS-4D. The IRIS-4D provides new cursor features: more colors, sizes and shapes. This chart compares the cursor features of the two workstations.

	2000/3000	4D
types	square	square and cross-hair
colors	1	3
size in pixels	16x16	16x16 and 32x32
hardware	drawn into bitplanes	handled by a special chip
writemask	cursor writemask	no writemask

On the IRIS 2000/3000, the cursor icon is written directly into one of the standard bitplanes. The data that the cursor icon replaces is stored in readily accessible raster memory. As the cursor moves across the screen, the system restores the data at the position from which it moves. It also saves the data for the area the cursor covers.

The display of the IRIS-4D cursor is controlled by special chips. The cursor is not stored in any of the bitplanes. Instead, for each pixel, the color is determined by multiplexing the cursor color against values found in the overlay, underlay or standard bitplanes. The cursor color always takes precedence over any color in another bitplane.

6.1 Porting Simple Cursors

On the IRIS 2000/3000, you can only create a 16x16 pixel cursor of a single color. `defcursor` loads the bit pattern for the cursor icon into a table of cursors. `setcursor` uses a cursor from the cursor table. `setcursor` requires three parameters: cursor table index, color, and writemask. This 2000/3000 code fragment creates a fly-shaped cursor.

```
static unsigned short  fly[16] = {
    0x0000, 0x1818, 0x2574, 0x2244,
    0x2184, 0xA185, 0xA185, 0xA185,
    0x518A, 0x2994, 0x1FF8, 0x15A8,
    0x2A54, 0x4BD2, 0x4422, 0x4002
};

writemask (0x7);

/* Load cursor into first entry of the cursor table. Use
 * curorigin() to center the "hot spot" (where getvaluator is
 * measured) of the cursor. In this example, the origin is
 * moved 8 units up and 8 units to the right.
 */
defcursor (1, fly);
curorigin (1, 8, 8);

/* If the window manager was not available, the cursor would
 * be drawn into the bitplane specified by the setcursor
 * subroutine. Since this program is written for the window
 * manager, the color and writemask (the last two parameters)
 * are ignored. In the 2000/3000 window manager, the color
 * of the cursor is determined by the .mexrc
 */
setcursor (1, 32, 32);
```

6.1.1 Using `defcursor`

On the IRIS-4D, `defcursor` is still used to load the cursor icon into the table. However, there is a new cursor drawing mode and color map to specify the color of the cursor.

Use `drawmode` for the cursor drawing mode and `mapcolor` to load color 1 in the cursor color map. Color 1 is used for the default 16x16 cursor. `drawmode` is only used to get colors for the cursor. Routines which affect the display of the cursor (`setcursor`, `curson`, and `cursoff`, for example) work, even when not in the cursor drawing mode.

There is no 4D cursor writemask, because the cursor is not written into a bitplane. Accordingly, `setcursor` needs only one parameter, the cursor table index.

```
defcursor (1, bitpattern);
drawmode (CURSORDRAW);
mapcolor (1, 255, 0, 0);
drawmode (NORMALDRAW);
setcursor (1);
```

This IRIS-4D code fragment defines and draws the same fly-shaped cursor shown earlier.

```
static unsigned short  fly[16] = {
    0x0000, 0x1818, 0x2574, 0x2244,
    0x2184, 0xA185, 0xA185, 0xA185,
    0x518A, 0x2994, 0x1FF8, 0x15A8,
    0x2A54, 0x4BD2, 0x4422, 0x4002
};

/* make a blue cursor in the shape of a fly.  Use defcursor()
 * to load the cursor table.  Load the blue RGB into the
 * cursor color map.  On the 3000, to activate the cursor,
 * setcursor() is called with three parameters:  cursor table
 * index, color and writemask.  The curorigin () routine
 * makes the "hot spot" (where the valuator is read from) the
 * middle of the cursor.  On the 4D, setcursor() takes only
 * one parameter: the entry in the cursor table.  The cursor
 * color has already been defined, and the 4D cursor has no
 * writemask.
 */
defcursor (1, fly);
curorigin (1, 8, 8);
drawmode (CURSORDRAW);
mapcolor (1, 0, 0, 255);
drawmode (NORMALDRAW);
setcursor (1);
```

Note: In FORTRAN, the `setcursor` subroutine is called `SETCUR`, and it still requires three arguments: the cursor table index, and two dummy arguments. In the above example, it would have this form:

```
CALL SETCUR (1, dummy, dummy)
```

6.2 Cross-hair Cursor

The cross-hair cursor consists of two intersecting lines that stretch horizontally and vertically from screen edge to screen edge. The cursor has no pattern. To define a cross-hair cursor, pass `defcursor` a null array for a bit pattern.

The cross-hair cursor can be drawn only in one color. Load the color for your cursor into **color 3** of the cursor color map.

For example, to make a pastel blue, cross-hair cursor, use this sequence of routines:

```
short *nofly;    /* dummy argument */
curstype (CCROSS);
defcursor (1, nofly);
drawmode (CURSORDRAW);
mapcolor (3, 100, 100, 255);
setcursor (1);
```

6.3 Cursors and The Window Manager

On the IRIS 2000/3000, depending upon whether the window manager is running, cursors behave differently. On the IRIS 2000/3000:

- the window manager is optional.
- the cursor color for the window manager is determined and fixed from the *.mexrc* file, when the window manager is initially activated. The color specified by `setcursor` calls is ignored, and the *.mexrc* cursor color is used. The color can only be changed if the program enters the pop-up menu mode: `pupmode()`.
- the window manager program controls the cursor, as long as the input devices are attached to the program. When the input devices become detached, the control of the cursor reverts to the window manager.

In contrast, on the IRIS-4D:

- the window manager is mandatory.

- the *.mexrc* file does not fix the cursor color. A graphics program uses a small cursor color map, so cursors can have several color entries.
- the window manager program controls the cursor, as long as the input devices are attached to the program. Cursor control to *attached* IRIS-4D programs is the same as the IRIS 2000/3000.

The window manager on both the IRIS 2000/3000 and IRIS-4D allows cursor control only for as long as input devices are attached to a program. For example, on either machine, if `cursoff` is used to stop display of the cursor, the cursor will reappear if you denote different states and actions of the window manager.

On the 2000/3000, the window manager cursor color is determined in the *.mexrc* file. On the 4D, the window manager cursor color is **color 1** in the cursor color map. This cursor color map is shared and controlled from graphics programs. Therefore, if a graphics program changes cursor color 1, all window manager cursors change color, as in the following sequence:

```
drawmode (CURSORDRAW);
mapcolor (1, red, green, blue);
```


7. RGB Mode Capabilities

On the IRIS-4D, the single and double buffer color map modes operate the same way as on the IRIS 2000/3000. In single buffer mode, up to 12 bitplanes are used to create a color index from 0 to 4095, which references a 4096 entry color map. In double buffer mode, up to 24 bitplanes are split into two equal-sized buffers. In color map mode, the value of the color index is stored directly into the standard bitplanes.

In RGB mode, the red, green, and blue components for a colored pixel are stored directly into the standard bitplanes. The color map is not referenced to determine the displayed color.

On the IRIS 2000/3000, RGB mode programs are limited to run only in single buffer mode and only without the window manager. On the IRIS-4D, RGB mode is supported in a wide variety of situations and its use is encouraged for real-time applications. On the 4D, RGB mode supports these features:

- window manager
- double buffer mode
- Gouraud shading
- depth cueing
- Z-buffering

For most applications, RGB mode is preferred to color map mode due to these advantages:

- More colors are available in RGB mode. Up to 16.7 million colors are simultaneously available in RGB mode, as compared to 4096 in color map mode. There are actually fewer pixels on the screen (1.3 million pixels) than RGB colors.

- RGB mode is more flexible for shading and depth cueing. For these operations in color map mode, a ramp of adjacent color indexes must be prepared. In RGB mode, the shading and depth cueing color values can be stored directly.

The only drawback to using RGB mode on the non-GT appears when you want to use double buffering. You have a small number of colors, and, depending on the application, may run more slowly than a double-buffered program that uses color map mode.

This table shows the contrasting RGB mode capabilities between the IRIS 2000/3000 and 4D workstations.

Display Mode	2000/3000	4D non-GT	4D/GT
single buffer	no RGB in 4Sight	8 bit RGB	24 bit RGB
double buffer	no RGB in 4Sight or double buffer mode	4 bit RGB	12 bit RGB

Table 7-1. RGB Mode in the Window Manager

7.1 RGB Mode in the Window Manager

On the 2000/3000, the window manager uses two standard bitplanes for each buffer and reduces the number of colors to one-fourth of the colors available without the window manager. A full complement of standard bitplanes are necessary for RGB mode on the 2000/3000, so RGB mode does not work in the window manager.

On the 4D, the window manager does not reduce the number of standard bitplanes, so RGB mode programs can run in the window manager. In single buffer mode, all 24 bitplanes comprise a full 24-bit RGB value, 8 bits for each color component. On the GT, these values are doubled.

7.2 Double Buffer and RGB Modes

The 4D also performs RGB mode and double buffering. On the non-GT 4D workstations, the 24 bitplanes are divided into two buffers with 12 bitplanes each. Each 12-bitplane buffer stores 4 bits for each color component (on the GT these values are doubled). The 4 bits create an 8-bit number by repeating the 4 bits in both the upper and lower bits in the 8-bit number. In double buffered RGB mode on the non-GT workstations, there are only 16 different shades for each color component. Colors appear banded as they change abruptly across smooth shaded and depthcued objects. The GT has enough bitplanes to overcome this problem.

7.3 Depth Cueing in RGB

You do not have to load a ramp of colors into the color map to depth cue in RGB mode. The graphics subroutine `RGBrange`, which is analogous to the subroutine `shaderange`, is introduced. The color map entries which contain the extreme dark and bright colors are passed to `shaderange`. For `RGBrange`, the actual red, green and blue component values are passed.

This code fragment demonstrates depth cueing in RGB mode.

```
setdepth (0xc000, 0x3fff);
perspective (400, 1.25, 100.0, 500.0);
translate (0.0, 0.0, -300.0);

/* This is analogous to the way depthcueing is done in color index
 * mode using,
 * shaderange(lowindex, highindex, zlow, zhigh)
 * RGBrange(rlow, glow, blow, rhigh, ghigh, bhigh, zlow, zhigh)
 */

depthcue (TRUE);
RGBrange (0, 0, 0, 255, 255, 0, 0xc000, 0x3fff);
move (-50.0, -50.0, -200.0);
draw (50.0, 50.0, 200.0);
depthcue (FALSE);
```

7.4 Supporting Multiple Display Modes

Many windows running different color display modes are simultaneously supportable. The multi-mode graphics processors (MGP) use data from the window bitplanes to determine whether to use the color map or RGB mode to interpret the data read from the image bitplanes. The MGPs provide three hardware mappings to control how the image bitplanes are read. Double buffer mode uses two mapping devices; single buffer mode uses only one.

When a window is opened, the mapping devices are examined to see if any are already being used to support its color display mode. If the new display mode is single buffer, it first tries to share a mapping device with a double buffer mode program of a similar color display mode. Failing in this, mapping devices are allocated to support the new display mode. If all mapping devices are used, some existing windows relinquish mapping devices for the new window. This makes the window, which relinquishes the mapping devices, appear with strange colors.

7.5 Other Features Working with RGB

Section 4.3 illustrates Gouraud shading in RGB mode. Section 5. Also, you can now use Z-buffering in RGB mode.

8. Multiple Windows Per Process

The window manager allows you to control several windows from a single graphics program. Each window has a unique *gid*. On the 4D, a maximum of sixteen windows is simultaneously displayable. On the 2000/3000, a single program was limited to ten windows, but far more than sixteen windows could be displayed at once.

The current graphics window is the window where drawing and window manipulation takes place. Only one graphics window is current for any program. To draw into a particular window, use the `winset (gid)` subroutine to make *gid* the current window.

Every window has its own matrix stack. You can modify the matrix stack for the current graphics window only.

All windows from a single program share the same event queue. When you attach the keyboard, mouse and other input devices to a single window, you are really attaching the devices to all the windows in that program.

If the REDRAW token is received on the queue, only the specified window really needs to be redrawn. The value which is passed along with the REDRAW token is the graphics identifier (*gid*) for the window you need to redraw. Use the `winset` subroutine to select that window, and redraw its contents.

```
device = qread (&val);
if (device == REDRAW) {
    winset (val);
    reshapeviewport ();
    drawscene ();
}
```

Also, in double-buffer mode, you must call `swapbuffers` for each window.

Porting Applications Update

This package contains the pages you need to update *Porting Applications to the IRIS-4D Family*, Version 1.0. Replace pages 6-9 through 6-12 of Section 1, Compatibility Guide, with the new pages provided.

A new cover and credits page are included to reflect the update. Replace the cover and credits pages. Your new cover and credits pages show a new version number. This new version number indicates that you have inserted the update package in your manual.

Make sure you remove the old version of the pages before you insert the new ones. Old pages have "Version 1.0" in the footer; the new ones have "Version 1.1" in the footer.

**Silicon Graphics, Inc.
2011 Stierlin Road
Mountain View, CA 94043**

Document Number 007-0909-011

Porting Applications to the IRIS-4D Family

Version 1.1

Document Number 007-0909-010
(Includes *Porting Applications Update*,
Document Number 007-0909-011)

Technical Publications:

Marcia Allen
Gail Kesner
Amy Smith
Diane Wilford

Engineering and Technical Marketing:

Kurt Akeley
Tom Davis
Gary Griffin
Todd Nordland
Mike Schulman
Thant Tessman
Mike Thompson
Chris Wagner
Mason Woo

© Copyright 1988, Silicon Graphics, Inc. - All rights reserved

This document contains proprietary and confidential information of Silicon Graphics, Inc., and is protected by Federal copyright law. The contents of this document may not be disclosed to third parties, copied or duplicated in any form, in whole or in part, without the express written permission of Silicon Graphics, Inc.

U.S. Government Limited Rights

Use, duplication or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (b) (2) of the Rights in Technical Data and Computer Software clause at 52.227-7013. Contractor/manufacturer is Silicon Graphics Inc., 2011 Stierlin Road, Mountain View, CA 94039-7311.

Porting Applications to the IRIS-4D Family**Version 1.1****Document Number 007-0909-010****(Includes *Porting Applications Update*,****Document Number 007-0909-011)**

Silicon Graphics, Inc.
Mountain View, California

The words IRIS, Geometry Link, Geometry Partners, Geometry Engine and Geometry Accelerator are trademarks of Silicon Graphics, Inc.

UNIX is a trademark of AT&T Bell Laboratories.

Allocating Memory to Share

To allocate memory to share between the driver and the application process, the driver map routine follows these steps:

1. Use the *kvpalloc* routine to allocate some pages in the kernel. Use the *btoc()* macro in *sys/sysmacros.h* to convert the *len* argument, which is in bytes, to pages.

```
kvpalloc(npages, flags)
int npages;
int flags;
```

kvpalloc allocates physical memory and returns a kernel virtual address associated with that memory. The physical memory is not subject to paging. The possible flags bits are defined in *sys/immu.h*. If *VM_UNCACHED* is set, *kvpalloc* returns an address whose data is not cached. If only one page is required, set *VM_DIRECT*, which causes *kvpalloc* to return a direct-mapped address—one that avoids the translation lookaside buffer hardware. If *VM_NOSLEEP* is set and there is no memory immediately available, *kvpalloc* returns *NULL*.

The operating system imposes a limit on the number of pages that can be locked down (not pageable). Note that *kvpalloc* can return *NULL* if there is not enough *lockable* memory available in the system. The driver can reserve the right to allocate locked pages by first calling the routine *reservermem()*.

```
int
reservermem(npages)
int npages;
```

If *reservermem* returns 0, you can call *kvpalloc* afterwards with the *VM_RESERVED* flag. In that case, the lockable-memory requirements are ignored. Note that passing *VM_RESERVED* without first reserving lockable pages can result in deadlocking the system.

2. Map the pages into the user's address space by using *v_mapphys*. The address argument, *addr*, is the virtual address returned by *kvpalloc*.
3. To free the memory, use *kvpfree*.

```
kvpfree(addr, npages)
caddr_t addr;
int npages;
```

Mapping the Contents of a Device

To map the contents of a device, the driver map routine calls the `v_mapreg` routine.

```
int
v_mapreg(vt, off, len)
vhandle_t *vt;
int off;
int len;
```

This routine maps the file associated with `vt`, the opaque handle to the user's virtual address space. The mapping begins at offset `off` for `len` bytes into the user's virtual address space associated with `vt`. The kernel passes these three arguments to the `drvmap` routine. Pages are faulted in on demand, and written to the device in these three cases:

- The user issues an `msync()` command.
- The virtual memory system chooses to reassign the page.
- The user unmaps the address space.

Mapping Device Registers

This section describes two ways to map device registers into the user's address space. The first way is to use the `v_mapphys` routine. The second way is for drivers that only want to map device registers, with no storage involved. In this case, you don't need to write a driver specifically for this purpose; instead, use a general interface via `/dev/mmem`, a special file associated with the general memory-mapping driver.

The `v_mapphys` routine sets up a virtual to physical mapping from `vt` to `addr`. `vt` is the opaque handle to the user's virtual address space (see previous section).

```
int
v_mapphys(vt, addr, len)
vhandle_t *vt;
caddr_t addr;
int len;
```

`addr` can be a physical I/O device address or a kernel address, but it cannot be a user virtual address. If a `k1seg` virtual address is used, or a physical

address that does not reference general memory, `v_mapphys` ensures that user references to the mapped space are not cached. The address must be page-aligned; the kernel supplies protections only on a page basis. `v_mapphys` returns 0 on success; it sets `errno` and returns -1 on failure.

Caution: Be very careful when you map device registers to a user process. Carefully check the range of addresses that the user requests to make sure that the request references only the requested device. Since protection is available only to a page boundary, configure the addresses of I/O cards so that they don't overlap a page. If they are allowed to overlap, an application process may be able to access more than one device, possibly a system device (for example, the disk or Ethernet). This is likely to cause problems.

The second way to map device registers is to use the general memory-mapping driver. Follow these steps:

1. Become the superuser.
2. Edit the file `/usr/sysgen/master.d/mem`. Add an entry in the array for *len* and *off*, where *len* is the size in bytes and *off* is the starting page-aligned physical address of the registers to be mapped.
3. Use *lboot* to reconfigure the set of mappable addresses. See the *lboot* man page and "Configuring a Kernel" in the *IRIS-4D Series Owner's Guide*.
4. When the user calls *mmap*, she or he passes a file descriptor associated with `/dev/mmem`, the special file associated with the general memory-mapping driver.

```
fd = open("/dev/mmem", O_RDWR);  
addr = mmap(0, len, prot, flags, fd, off);
```

len and *off* are the entries that you made in the array in `/usr/sysgen/master.d/mem`.

Returning Data Associated with the Opaque Handle

To return the unique identifier associated with *vt*, the opaque handle to the user's virtual address space, use the *v_gethandle* command.

```
unsigned  
v_gethandle(vt)  
vhandle_t *vt;
```

Since the virtual handle points into the kernel stack, it is likely to be overridden. Use *v_gethandle* if your driver must “remember” several virtual handles.

To return the virtual address in the process where the space is attached, use the *v_getaddr* command.

```
caddr_t  
v_getaddr(vt)  
vhandle_t *vt;
```

To return the length in bytes of the virtual space, use the *v_getlen* command.

```
int  
v_getlen(vt)  
vhandle_t *vt;
```